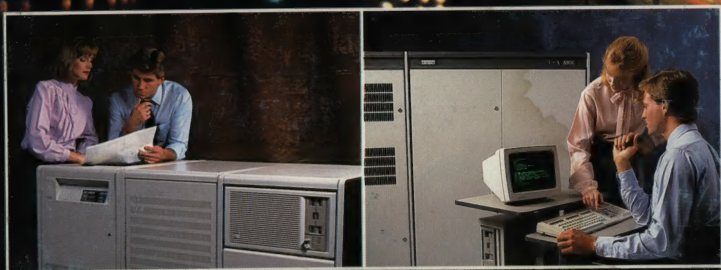


digital

VAX Architecture Handbook





VAX Architecture Handbook

1986

digital

Digital believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. Digital is not responsible for any inadvertent errors.

The following are trademarks of Digital Equipment Corporation:

DEC	PDP	RT
DECmate	P/OS	UNIBUS
DECUS	Professional	VAX
DECwriter	Rainbow	VMS
DIBOL	RSTS	VT
MASSBUS	RSX	Work Processor
MicroVAX		

And the Digital Logo: **digital**

Copyright © 1986 Digital Equipment Corporation. All Rights Reserved.

CONTENTS

Preface

Chapter 1 ■ VAX Architecture Design

Introducing VAX Architecture	1-1
Virtual Address Extension	1-2
PDP-11 Computer Compatibility Option	1-3
Memory Management	1-3
Additional Literature	1-4

Chapter 2 ■ VAX Architecture Overview

Processing Concepts	2-1
Context Switching	2-2
Priority Dispatching	2-3
Virtual Addressing	2-4
Memory Management	2-4
Instruction Set	2-5
Routine Call Capability	2-7
Instruction Operand Processing	2-7
Process Control Instructions	2-8
General Register Addressing	2-9
Data Types	2-10
Stacks	2-13
Condition Codes	2-13
Exceptions and Interrupts	2-14
Input/output Control	2-15

Chapter 3 ■ VAX Architecture Characteristics

Programming Environment	3-1
Processor Status Longword	3-1
Processor Access Modes	3-1
Special Instructions	3-2
Data Sharing	3-3
Data Access Synchronization	3-3

Registers.....	3-4
General Registers.....	3-4
Processor Registers.....	3-7
Input/output Registers.....	3-13
Stacks.....	3-14
Cache Memory.....	3-16
Restartability.....	3-17
Interrupts and Errors.....	3-18

Chapter 4 ■ Data Representation

Character String Data.....	4-1
Floating-point Data.....	4-2
D_ floating Data.....	4-2
F_ floating Data.....	4-3
G_ floating Data.....	4-4
H_ floating Data.....	4-5
Integer Data.....	4-6
Byte Data.....	4-7
Word Data.....	4-7
Longword Data.....	4-8
Quadword Data.....	4-8
Octaword Data.....	4-9
Numeric String Data.....	4-10
Leading Separate Numeric String Data.....	4-10
Trailing Numeric String Data.....	4-12
Packed Decimal String Data.....	4-16
Queue Data.....	4-17
Variable Length Bit Field Data.....	4-21
Data in Registers.....	4-24

Chapter 5 ■ The Instruction Characteristics

Notation Convention.....	5-1
Assembler Notation.....	5-1
Operand Notation.....	5-1
Operation Notation.....	5-2
Range and Extent Notation.....	5-5
MACRO Source Statement Format.....	5-5
Instruction Format.....	5-6
Operator Field.....	5-8
Operand Field.....	5-9

Addressing Modes.....	5-10
General Mode Addressing.....	5-12
Branch Mode Addressing.....	5-41

Chapter 6 ■ Functions of the Instruction Set

Address Instructions.....	6-1
Arithmetic Instructions.....	6-2
Character String Instructions.....	6-2
Control Instructions.....	6-3
Case Instructions.....	6-4
Loop Control Instructions.....	6-4
Subroutine Call Instructions.....	6-4
Transfer Instructions.....	6-5
Cyclic Redundancy Check Instruction.....	6-5
Decimal String Instructions.....	6-6
Edit Instruction.....	6-9
Floating-point Instructions.....	6-10
Index Instruction.....	6-12
Integer Instructions.....	6-13
Logic Instructions.....	6-13
Multiple Register Instructions.....	6-14
Privileged Instructions.....	6-15
Procedure Call Instructions.....	6-16
Processor Status Longword Instructions.....	6-18
Queue Instructions.....	6-19
Absolute Queue Instructions.....	6-19
Self-relative Queue Instructions.....	6-22
Variable Length Bit Field Instructions.....	6-25

Chapter 7 ■ Memory Management

Virtual Memory.....	7-2
Virtual Address Space.....	7-6
Address Translation.....	7-8
Page Table Entry.....	7-8
Page Table Entry for I/O Devices.....	7-10
Changing Page Table Entries.....	7-11
System Space Address Translation.....	7-11
Process Space Address Translation.....	7-13
Access Control.....	7-16

Controlling Memory Management	7-19
Faults and Parameters	7-20
Accessing Privileged System Services	7-22

Chapter 8 • Exceptions and Interrupts

Event Handling	8-1
Interrupt Priority Levels	8-3
Exceptions and Interrupts	8-3
Processor Status	8-4
Asynchronous System Traps	8-7
Exceptions	8-9
Arithmetic Exceptions	8-9
Instruction Fault	8-11
Memory Management Exceptions	8-12
Operand Reference Exceptions	8-12
Serious System Failures	8-14
Trace Exceptions	8-15
Interrupts	8-18
Device Interrupts	8-19
Software-generated Interrupts	8-19
Urgent Interrupts	8-21
Interrupt Priority Level Register	8-21
Interrupt Example	8-22
System Control Block	8-23
Stacks	8-27
Stack Location	8-28
Stack Alignment	8-28
Status Bits	8-29
Accessing Stack Registers	8-30
Recognition Priority	8-30
Suspended Instructions	8-31
Initiating an Exception or Interrupt	8-31

Chapter 9 • The Instruction Set

Add	9-1
Add Aligned Word Interlocked	9-2
Add Compare and Branch	9-2
Add One and Branch	9-3
Add Packed	9-3
Add with Carry	9-4
Arithmetic Shift	9-4

Arithmetic Shift and Round Packed	9-4
Bit Clear	9-5
Bit Clear Processor Status Word	9-5
Bit Set	9-6
Bit Set Processor Status Word	9-6
Bit Test	9-6
Branch	9-7
Branch on Bit	9-7
Branch on Bit Interlocked	9-7
Branch on Bit and Modify without Interlock	9-8
Branch on <i>Condition</i>	9-8
Branch on Low Bit	9-10
Branch to Subroutine	9-11
Breakpoint Fault	9-11
Bugcheck	9-11
Call Procedure with General Argument List	9-12
Call Procedure with Stack Argument List	9-13
Case	9-14
Change Mode	9-14
Clear	9-15
Compare	9-15
Compare Characters	9-16
Compare Field	9-17
Compare Packed	9-17
Convert	9-18
Convert Leading Separate Numeric to Packed	9-20
Convert Longword to Packed	9-20
Convert Packed to Leading Separate Numeric	9-21
Convert Packed to Longword	9-21
Convert Packed to Trailing Numeric	9-22
Convert Trailing Numeric to Packed	9-23
Cyclic Redundancy Check Instruction	9-23
Decrement	9-26
Divide	9-26
Divide Packed	9-27
Edit Instruction	9-27
EO\$ADJUST__ INPUT	9-29
EO\$BLANK__ ZERO	9-29
EO\$CLEAR__ SIGNIF	9-29
EO\$END	9-30
EO\$END__ FLOAT	9-30
EO\$FILL	9-30
EO\$FLOAT	9-31

EO\$INSERT.....	9-31
EO\$LOAD	9-32
EO\$MOVE.....	9-32
EO\$REPLACE__ SIGN.....	9-33
EO\$SET__ SIGNIF.....	9-33
EO\$STORE__ SIGN.....	9-34
Exclusive OR	9-34
Extended Divide	9-35
Extended Function Call	9-35
Extended Modulus.....	9-35
Extended Multiply.....	9-36
Extract Field.....	9-36
Find First Bit	9-37
Halt	9-37
Increment.....	9-38
Index.....	9-38
Insert Entry in Queue.....	9-38
Insert Entry into Queue at Head, Interlocked	9-39
Insert Entry into Queue at Tail, Interlocked	9-40
Insert Field.....	9-41
Jump	9-41
Jump to Subroutine	9-41
Load Process Context	9-42
Locate Character	9-42
Match Characters	9-43
Move.....	9-43
Move Address	9-44
Move Characters	9-44
Move Complement	9-45
Move from Processor Register	9-45
Move from Processor Status Longword.....	9-47
Move Negated.....	9-47
Move Packed	9-48
Move to Processor Register	9-48
Move Translated Characters	9-49
Move Translated until Character.....	9-49
Move Zero-extended	9-50
Multiply.....	9-50
Multiply Packed	9-51
Polynomial Evaluation	9-51
Pop Registers	9-52
Probe Accessibility.....	9-52
Push Address	9-53

Push Longword.....	9-54
Push Registers.....	9-54
Remove Entry from Queue.....	9-54
Remove Entry from Queue at Head, Interlocked.....	9-55
Remove Entry from Queue at Tail, Interlocked.....	9-56
Return from Exception or Interrupt.....	9-56
Return from Procedure.....	9-57
Return from Subroutine.....	9-58
Rotate Longword.....	9-58
Save Process Context.....	9-58
Scan Characters.....	9-59
Skip Character.....	9-59
Span Characters.....	9-60
Subtract.....	9-60
Subtract One and Branch.....	9-61
Subtract Packed.....	9-62
Subtract with Carry.....	9-62
Test.....	9-63

Chapter 10 • Architectural Subsetting

Subsetting Rules.....	10-1
Floating-point Instructions.....	10-1
String Instructions.....	10-2
Compatibility Mode Instruction Set.....	10-2
Processor Registers.....	10-2
The Kernel Instruction Set.....	10-3
Instruction Emulation.....	10-4
MicroVAX I Systems.....	10-4
MicroVAX II Systems.....	10-4

Chapter 11 • PDP-11 Compatibility Mode

PDP-11 User Environment Emulation.....	11-2
General Registers.....	11-2
Stack Pointer Register.....	11-3
Processor Status Word.....	11-3
Compatibility Mode Instructions.....	11-4
Entering and Leaving PDP-11 Compatibility Mode.....	11-6
Memory Management.....	11-7
Exceptions and Interrupts.....	11-10
Tracing in Compatibility Mode.....	11-10

Unimplemented PDP-11 Traps	11-11
Input/output References	11-11
Processor Registers	11-11
Program Synchronization	11-11

Appendix A • Powers of Binary and Hexadecimal Numbers	A-1
--	------------

Appendix B • List of Instructions by Mnemonic	B-1
--	------------

Appendix C • List of Instructions by Opcode	C-1
--	------------

Glossary	Glossary-1
-----------------------	-------------------

Index	Index-1
--------------------	----------------

Preface

The primary purpose of this handbook is to provide the detail needed to make a sound technical evaluation of the capabilities and characteristics of the VAX Architecture. A secondary purpose is its use as a text for students of computer architecture. The handbook is not an assembly language reference. However, readers interested in assembly language programming will find this handbook an excellent introduction to that subject.

1875. The year of the great drought. The crops were
 very poor. The people were very poor. The
 year was a very bad year for the people.

Chapter 1 ■ VAX Architecture Design

During the next decade, computers and the computer industry will witness ever-increasing, perhaps unpredictable, demands. In finance, government, industry, and in the home, computers will serve expanding roles, solving problems, managing processes, or facilitating communication. Digital developed an innovative computer technology to confront these challenges—a technology that offers vast power and enormous flexibility for every application. At the same time, we have held fast to the philosophy of affordability and ease of use that made Digital the leader of the minicomputer industry.

Scientific, industrial, commercial, and educational market users have already put the original VAX model through its paces in numerous situations—real-time, computational, program development. In the coming decade we will see a wide range of new tasks handled by VAX processors.

■ Introducing VAX Architecture

The VAX architecture is the heart of the VAX processor family. We define architecture as the collection of attributes common to all VAX processors—attributes that guarantee that all software developed on a VAX processor runs without change on all VAX processors.

Particularly pertinent attributes are the instruction set, memory management, and certain other aspects of the design that help define contexts and processes. Let us make a distinction between the architecture and the implementation of that architecture. For example, the architecture of the typewriter is essentially fixed: it is the keyboard layout. With knowledge of the alphabet and punctuation systems, any typist can make a typewriter work—can process jobs. However, each manufacturer may implement that architecture in differing ways. Some may have striking print keys while others may have spherical typing elements. Some may have a blue keyboard, some black. In addition, the manufacturer could trade one feature for another; for example, a lighter touch versus the capability to make a number of carbons. Nevertheless, all typewriters still perform an essential function, typing.

Computer architectures also perform an essential function. Each processor in the family may bear slightly differing implementations and tradeoffs. Yet all will fulfill the requirements of the machine. And all will deliver the same service to the users. For example, having learned the instruction set, a programmer is ensured that an instruction performs precisely the same operation on each processor in the VAX computer family. This includes the MicroVAX processors that use a subset of the VAX instruction set.

VAX architecture is appropriate over a variety of system costs, performance and application needs. Therefore, a broad range of user requirements can be met at a lower cost because the price of supporting many different architectures is eliminated.

The most visible attribute of VAX architecture is the instruction set. Over three hundred instructions give an assembly-level programmer extensive control of computer operation. Each instruction has a mnemonic, a shorthand name that suggests its function. (Obvious mnemonics are ADD, DIV, MOV, and PUSH.) Independence is incorporated into the instruction set. That is, the operation being performed, the type of data used, and the method of addressing can all be considered independently by the compiler. This makes for faster, more efficient, and easier to implement compilers.

Some recurrent operations from high-level languages are engineered into the hardware so that a single instruction can handle them. The FORTRAN DO loop and three-operand addition ($A = B + C$) are examples of operations that are handled by a single VAX instruction.

The instructions include provisions to make various applications and operating system codes more efficient. There are, in this group, hardware support of queues, easy access to bit fields of variable lengths, and simple instructions to save or restore a program context.

Because Digital foresaw the possibility of adding more and more applications, the instruction set is extendible. The instruction set can be expanded to include new data types and operators in a way that consistently matches all the ones that already exist. Enormous flexibility is assured this way, because what exists now does not significantly constrain what may be added in the future.

■ Virtual Address Extension

The word VAX suggests the premier feature of VAX processors—virtual address extension. In a VAX computer, information is located with a 32-bit address. This means effectively that the computer can recognize more than four billion addresses. In minicomputer and programming terms, this is an enormous address range. The remarkable thing about this address space is that it is *virtual*.

The physical memory of the computer need not be nearly as large as the four billion bytes for the machine to process data whose addresses are scattered through the address space. In fact, what happens is that a sophisticated scheme called *memory management* allows programmers to operate as if a major part of the virtual address space is available to them. Memory management handles all the details of storing programs and subsequently bringing them into main memory where they are processed.

From the programmers' points of view, two billion bytes of virtual address space can be used for programs. Programmers need never worry about the complicated techniques of overlaying or segmenting to squeeze the program into a smaller address range. Logic is built into the VAX processors to quickly

-
- translate all the virtual addresses to physical addresses
-
- store the programs and data in convenient locations
-
- bring into main memory whatever parts of the program or data are needed at any instant.
-

Another characteristic of the VAX architecture is the rapid switching of process context. VAX machines are high-powered processors. Many programs and many programmers can use a VAX processor simultaneously, with each appearing to have control. Actually, the processor is executing pieces of one program and switching back and forth to execute other programs. A *switched-in context* allows a program to run. A *switched-out context* makes the program wait for the processor. Consequently, many different activities can occur on a VAX processor at any one time. Context switching takes place so quickly that no one is aware of the change.

▪ PDP-11 Computer Compatibility Option

We use the word *compatibility* to designate VAX systems' connection to the PDP-11 computers. Customers have a large investment in the PDP-11 computers and software. To protect that investment, and to simplify the migration procedures, Digital offers optional software to ensure that VAX systems accept with minimal conversion many types of PDP-11 programs.

Conversely, VAX systems offer an excellent host development environment for applications that will eventually run on PDP-11 computers. Naturally, there are some restrictions. But most of the time, a simple recompilation of programs is all that is required to carry a PDP-11 program to a VAX processor. Compatibility mode programs may execute with native mode programs in a VAX system environment.

▪ Memory Management

The memory management hardware is responsible for maintaining virtual memory environment and for enforcing memory protection between access modes. But that is only a part of the memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment.

Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware translates a virtual address into a physical address.

NOTE

A physical address is the address exchanged between the processor, memory, and the peripheral adapters. Typically, the physical address is transparent to the programmer, who deals with virtual addresses.

■ Additional Literature

Additional literature on the VAX architecture is available. The literature is directed toward two types of readers—those who need only enough detail to evaluate the VAX processors and those who need the myriad of detail needed to develop assembly language programs. Should you need for that literature, it can be ordered through your local Digital Sales Office or through the Digital Accessories and Supplies Group.

For the evaluators, there is the VAX Handbook Series of which this handbook is one. They cover a variety of subjects related to VAX processors—hardware, software, languages, tools, and others.

For assembly language programmers (and in-depth evaluators), there is a manual available that describes the architecture in great detail (VAX-11 Architecture Reference Manual, Order Code EK-VAXAR-RM-002). The manual provides a functional description of the behavior of the VAX processors. In addition, there is a programming card containing the instructions in tabular form. The card lists by opcode each instruction, its arguments, and the affect the instruction has on the condition codes (VAX-11 Programming Card, order number AV-D827C-TE).

We hope that the handbooks answer most of your questions about the VAX family of computers, their architecture, and the abundance of available software. If you have more questions, your Digital sales representative will be happy to help you.

Chapter 2 ■ VAX Architecture Overview

The term *VAX architecture*, when used in the context of this discussion, refers to the functional behavior of a VAX processor as opposed to the logical design and the physical implementation. The primary advantage of a common family architecture is that it provides the ability to create software on one processor and execute that software on any other processor in the family.

NOTE

For your convenience, this handbook contains a glossary of words and terms that have either a unique meaning in VAX systems or are used with special meaning.

■ Processing Concepts

VAX processors are designed specifically to support a high-performance multiprogramming environment. The major advantage of a multiprogramming system is its ability to share its resources. For example, multiprogramming enables the apparently simultaneous execution of many applications and the interactive development of applications programs. Hardware characteristics that support multiprogramming are

-
- Rapid context switching.
 - Priority dispatching.
 - Virtual addressing.
 - Memory management.
-

Multiprogramming VAX systems not only share the processor among several processes but also protect processes from one another while allowing them to communicate and share both code and data.

A process is the basic entity that can be executed by a VAX system. Processes consist of an address space, a hardware context, and a software context. The hardware context is defined by a process control block (PCB). The block is a data structure containing images of the general purpose registers, processor status longword, program counter, process stack pointers, process mapping registers, and several minor control fields.

When a process is not executing, its hardware context is stored in the process control block. Most of the process control block must be moved to internal registers for the process to execute. When a process is executing, some of its hardware context is updated in the internal registers.

Saving the contents of the privileged registers in the process control block of the currently executing process and then loading a new set of context in the privileged registers from another process control block is termed context switching. Context switching occurs as one process after another is scheduled for execution.

Context Switching

In a multiprogramming environment, several individual streams of code can be ready to execute at any time. Instead of allowing each stream to execute to completion serially (as in a batch-only stream), the operating system intervenes and switches among them. In VAX computers, the hardware establishes an environment for rapid switching. Switching occurs to increase the efficiency of the computer by using its resources in a balanced fashion, and to allow the intervention of processes or events that require priority treatment.

The stream of code executing at any one time is determined by its hardware context; that is, the information that is in processor registers. That information identifies

-
- The location of the stream's instructions and data.
-
- Which instruction to execute next.
-
- The processor status during execution.
-

Therefore, a *process* is a stream of instructions and data defined by a hardware context. Each process has a unique identification. The operating system switches between processes by requesting the processor to save one process's context and load another. Context switching occurs rapidly because the instruction set includes instructions that save and load hardware context.

For each process eligible to execute, the operating system creates a data structure called the software process control block. Within that block is a pointer to another data structure, the hardware process control block. That control block contains the hardware process context, that is, all the data needed to load the processor's programmable registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor's process control block base register with the physical address of a hardware process control block and issues the *load process context* instruction. The processor loads the process context in one operation and is ready to execute code within that context.

The process control block also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps (AST) to processes. The AST field is used to schedule a software interrupt. The interrupt initiates an AST routine and ensures that they (interrupt and AST routine) are delivered to the proper process.

Priority Dispatching

While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory, and peripheral resources among many processes, the processor has two arbitration mechanisms that support high-performance multiprogramming—exceptions and interrupts. Exceptions are events that occur synchronously (predictably) with respect to the execution of a particular stream of instructions. Interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. They are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

Systemwide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines. Systemwide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.

Systemwide changes in flow take priority over process-local ones. Furthermore, the processor uses a priority system for servicing interrupts. Each kind of interrupt is assigned a priority, and the processor responds to the highest-priority interrupt pending. For example, interrupts from the high-speed disk devices take precedence over interrupts from low-speed devices.

The processor services interrupts between instructions, or at well-defined points during the execution of long, iterative instructions. When the processor acknowledges an interrupt, it switches rapidly to a special systemwide context to enable the operating system to service the interrupt. Systemwide changes in the flow of execution are handled so they do not disrupt individual processes.

Virtual Addressing

Most data is located in memory using the address of an 8-bit byte. Virtual addresses identify the byte locations. Such addresses are called virtual because they are not the real addresses for physical memory locations. Rather, they are translated into real addresses by the processor under operating system control.

A virtual address, unlike a physical memory address, is not a unique address of a location in memory. For example, two programs using the same virtual address might refer to two different physical memory locations. Conversely, two programs could refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. It can be viewed as an array of byte locations labeled from 0 to 4,294,967,295 ($2^{32} - 1$). This space is divided into sets of virtual addresses designated for certain uses: those used by processes constitute half of the total virtual address space, and are collectively designated as process space. Addresses in the remaining half of virtual address space refer to locations maintained and protected by the operating system, and are collectively designated as system space.

Memory Management

Memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. The memory management hardware oversees the handling of virtual address space including memory protection.

Virtual address space is divided into pages. Each page represents 512 bytes of contiguously addressed memory. The first page begins at byte 0 and continues to byte 511. The next page begins at byte 512 and continues to byte 1023, and so forth.

To make memory mapping efficient, the processor must be able to translate virtual addresses rapidly to physical addresses. Two features providing rapid address translation are the processor's internal address translation buffer and the translation algorithm itself.

The processor has three pair of page mapping registers. Two pair are for the process space (P0 and P1) and one pair for system space. The operating system's memory management software loads the pairs of registers with base addresses and lengths of data structures called page tables. The tables provide the mapping information for each virtual page in the system. Thus, there is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each entry is a longword representing the physical mapping and protection for one virtual page. To translate a virtual address to a physical address, the processor uses the virtual page number as an index into the page table from the given page table base address. Each translation contains 512 virtual addresses.

All process page tables have virtual addresses in the system region of virtual address space. But the system region page table itself is located by its address in physical memory. That is, the system region page table base register contains the physical address of the page table base, while the process page table base registers contain the virtual addresses of their page table bases.

There are two advantages to using a virtual address as the base address of a process page table. The first is that all page tables need not reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. The process page tables can reside in auxiliary memory. That is, they can themselves be paged and swapped as necessary.

The second advantage is that the operating system's memory management software can allocate process page tables dynamically because they do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes and therefore does not require redefinition from context to context.

Memory protection is implemented by having four access modes. Each process is assigned an access mode. The hardware checks the memory access request against the assigned access mode. There are four access modes: kernel, executive, supervisor, and user. The kernel mode has the highest degree of access while the user has the lowest degree of access. Memory management is described in greater detail later in this book.

▪ Instruction Set

A major goal of the VAX architecture is to provide an instruction set that is symmetrical with respect to data types. Symmetrical operations include data movement, data conversion, data testing, and computation. Thus, the best instruction for the data type can be selected for optimum processing.

Instruction mnemonics are formed by combining an operator abbreviation with a data-type suffix. Conversion instructions are formed by adding suffixes for both the source and destination data types. Computation instructions have an additional suffix to indicate the choice between two-operand and three-operand instructions. Instruction mnemonics were carefully chosen to ensure they perform the task for which they were designed.

A native-mode instruction may start on any byte boundary. The variable-length instruction format makes code more compact and also guarantees that the instruction set can be easily extended. Operation codes or *opcodes* for the operations are single or double bytes followed by up to six operand specifiers. An operand specifier can be from 1 to 17 bytes long depending on the addressing mode and data type.

Figure 2-1 illustrates the autodecrement mode *move longword* instruction as a string of bytes starting with the opcode followed by two operand specifiers. In this example, the assumed starting location is 00003000. When the processor completes the execution of an instruction, the program counter contains the address of the first byte of the next instruction. Program counter operation is totally transparent to the programmer.

MACHINE CODE: (ASSUMED STARTING LOCATION 00003000)

00003000	D0	OPCODE FOR MOVE LONG INSTRUCTION
00003001	73	AUTODECREMENT MODE, REGISTER R3
00003002	54	REGISTER MODE, REGISTER R4

Figure 2-1 ■ Autodecrement Move Longword Instruction

The program counter can be used to identify operands. The assembler translates many types of operand references into addressing modes using the program counter. The addressing modes have names different from those when other registers are used. When using the program counter in autoincrement mode, the mode is called immediate mode. Immediate mode is used to specify inline constants. Autoincrement deferred mode using the program counter is called absolute mode. Absolute mode is used to reference an absolute address. Displacement and displacement-deferred modes using the program counter are used to specify an operand using an offset from the current location.

Addressing using the program counter enables the coding of position-independent code. Position-independent code can be executed anywhere in virtual address space after it has been linked. Program linkages are identified as absolute locations in virtual address space. All other addresses are identified relative to the current instruction.

Routine Call Capability

The processor provides two kinds of routine call instructions—those for subroutines and those for procedures. In general, a subroutine is a routine entered using a *jump to subroutine* or *branch to subroutine* instruction, while a procedure is a routine entered using a *call* instruction.

The processor automatically saves and restores the contents of registers to be preserved across procedure calls, and it provides two methods for passing argument lists to called procedures—by passing the arguments on the stack and by passing addresses of arguments elsewhere in memory. The processor also constructs a *journal* of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure's stack data, called its *stack frame*, enables proper returns from procedures even when the procedures leave data on the stack. In addition, user and operating system software can *unwind* the stack frame to trace back through nested calls to handle errors or debug programs.

Instruction Operand Processing

The following three steps are performed by each instruction during execution. First, each operand specifier is evaluated by type of access in the order in which they appear in the instruction stream.

1. Read access—evaluate the operand address, read the operand, and save the operand.
2. Write access—evaluate the operand address and save the address.
3. Modify access—evaluate the operand address, read the operand, save both the address and the operand.
4. Address and branch access—evaluate the address and save the address.
5. Field access—evaluate the operand base address and save the address.

Second, the operation indicated by the instruction is performed. Third, the result or results are stored using the saved address in the order indicated by the occurrence of operand specifiers in the instruction stream.

NOTE

Character and numeric string instructions write any output strings and store the registers during step 3.

The implications of this processing are

1. Autoincrement and autodecrement operations occur as the operand and specifiers are processed and subsequent operand specifiers use the updated contents of register modified by those operations.

2. Except for those operations mentioned in step 1, all input operations are read and all addresses of output operands are computed before any results of the instruction are stored.
3. An operand of modify access type is not read, modified, and written as an indivisible operation. Therefore, modify access type operands cannot be used for synchronization. For synchronization, refer to the ADAWI, BBCCI, BBSSI, INSQHI, INSQTI, REMQHI, and REMQTI instructions.
4. If an instruction references two operands of write or modify access type at the same or overlapping address, the first will be overwritten by the second. If an instruction modifies a register implicitly and also has an output operand, the output store is performed after the register update.

Process Control Instructions

Process scheduling software executes on the interrupt stack. This protocol makes available a noncontext-switched stack. If the scheduler were running on a process's kernel stack, any state information in that stack would disappear whenever a new process is selected. Running on the interrupt stack can occur as the result of the interrupt origin of scheduling events. Some synchronous scheduling requests (such as a WAIT service) may cause rescheduling without any interrupt occurrence. For this reason, the *save process context* (SVPCTX) instruction can be executed while on either the kernel or interrupt stack. It forces a transition to execution on the interrupt stack.

All of the process control instructions are privileged and can be executed in kernel mode only. Example 2-1 illustrates how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatch routine is always entered by way of an interrupt.

Example 2-1 ■ Simple Dispatch Routine

```

; ENTERED VIA INTERRUPT, IPL=3

RESCHED: SVPCTX                ; Save context in PCB.

    .
    .
    .
    (set state to runnable and place current PCB on proper RUN queue)
    .
    .
    .
    (Remove head of highest priority, nonempty, RUN queue)
    MTPR @#PHYSPCB, PCBB      ; Set physical PCB address in PCBB.
    LDPCTX                    ; Load context from PCB for new process.
    REI                        ; Place process in execution.

```


▪ General Register Addressing

Within the processor there are locations called general registers that can be used for temporary data storage and addressing. Sixteen 32-bit general registers are available for use with the native instruction set, though some have special significance. For example, one register is designated as the program counter, and it contains the address of the next instruction to be executed.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The method by which an operand location is specified is called the operand addressing mode. VAX processors offer a variety of addressing modes and addressing mode optimizations: one addressing mode locates an operand in a register; several other addressing modes locate an operand in memory by using a register to point to the operand, point to a table of operands, or point to a table of operand addresses.

Other addressing modes exist that are indexed modifications of the addressing modes that locate an operand in memory. Finally, still other addressing modes identify the location of the operand in the instruction stream, including one for constant data and one for branch instruction addresses. The general register addressing modes are briefly summarized in Table 2-1.

Table 2-1 ▪ General Register Addressing Modes

Mode	Symbol	Indexed
Absolute	@# <i>addr</i>	[R _x]
Autodecrement	– (R _n)	[R _x]
Autoincrement	(R _n) +	[R _x]
Autoincrement Deferred	@(R _n) +	[R _x]
Displacement—		
Byte	B↑ <i>displ</i> (R _n)	[R _x]
Word	W↑ <i>displ</i> (R _n)	[R _x]
Longword	L↑ <i>displ</i> (R _n)	[R _x]
Displacement Deferred—		
Byte	@B↑ <i>displ</i> (R _n)	[R _x]
Word	@W↑ <i>displ</i> (R _n)	[R _x]
Longword	@L↑ <i>displ</i> (R _n)	[R _x]
Immediate	I↑# <i>constant</i>	NA
Literal	S↑# <i>constant</i>	NA
Register	R _n	[R _x]
Register Deferred	(R _n)	[R _x]

Legend: $n = 0:15$, $x = 0:14$, *displ* = displacement address

▪ Data Types

The processor's native instruction set recognizes several primary data types—character-string, floating-point, integer, numeric-string, packed-decimal, and variable-length bit field. For each of these data types, the selection of operation immediately informs the processor of the size and interpretation of the data. This is done so that the processor can then manipulate the bit field as a function of user-defined field size and relative position from a given byte address.

Several variations of the primary data types exist. Table 2-2 provides a summary of all the data types available. Figure 2-2 illustrates some of them graphically.

Table 2-2 • Data Types

Data Type	Size	Range (decimal)
Integer—		Signed Unsigned
Byte	8 bits	– 128 to + 127 0 to 255
Word	16 bits	– 32768 to + 32767 0 to 65535
Longword	32 bits	– 2^{31} to + $2^{31} - 1$ 0 to $2^{32} - 1$
Quadword	64 bits	– 2^{63} to + $2^{63} - 1$ 0 to $2^{64} - 1$
Octaword	128 bits	– 2^{127} to + $2^{127} - 1$ 0 to + $2^{128} - 1$
Floating Point—		
D__ floating	64 bits	approximately 16 decimal digits precision
F__ floating	32 bits	approximately 7 decimal digits precision
G__ floating	64 bits	approximately 15 decimal digits precision
H__ floating	128 bits	approximately 33 decimal digits precision
Packed Decimal String	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte
Character String	0 to 65535 bytes	one character per byte
Variable-length Bit Field	0 to 32 bits	dependent on interpretation
Numeric String	0 to 31 bytes (digits)	– $10^{31} - 1$ to + $10^{31} - 1$
Queue	≥ 2 longwords/queue entry	0 through 2 billion entries

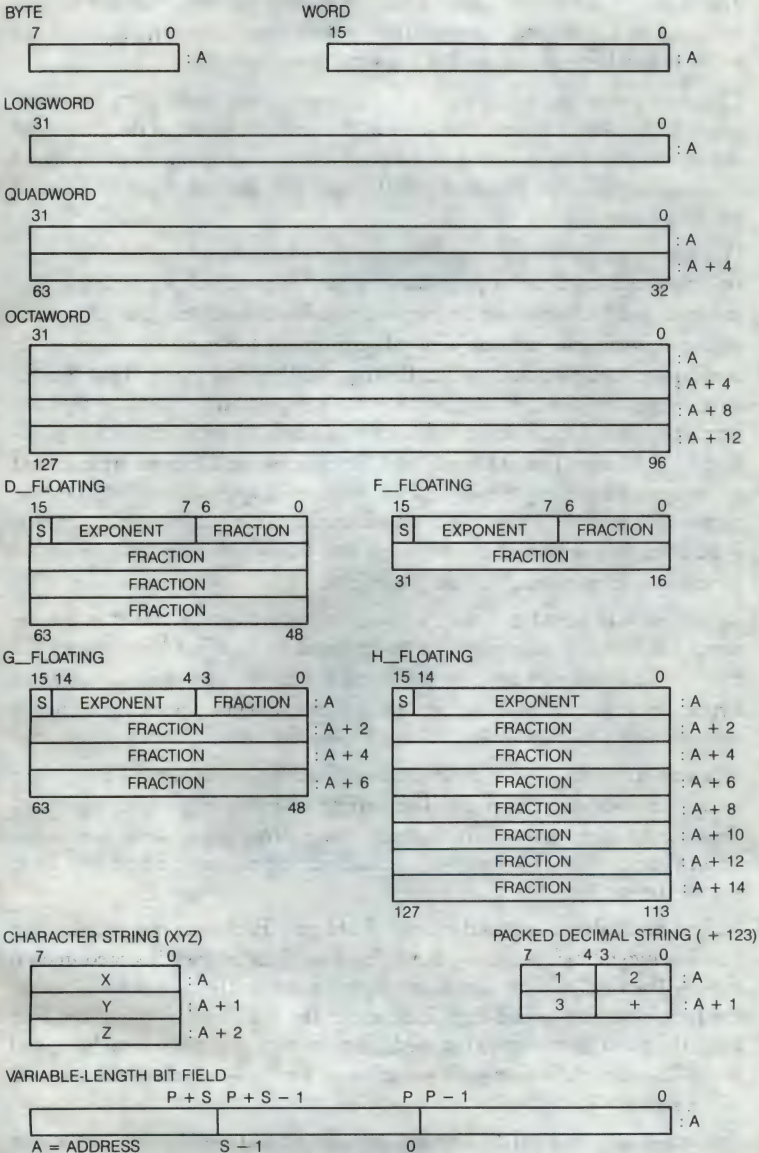


Figure 2-2 ■ Data-type Representations

The data type of an instruction operand identifies how many bits of storage should be considered as a unit and what is to be the interpretation of that unit. This is important because, as you will see in later sections, identical bit patterns can be interpreted as very different data items; similarly, different bit patterns may be used to represent the same numerical value.

Character string data is a string of bytes containing any binary data, for example, ASCII codes. The first character in the string is stored in the first byte, the second character is stored in the second byte, and so on. In particular, a character string that contains ASCII codes for decimal digits is called a numeric string.

Floating-point values are stored using a signed exponent and a binary fraction. Four types of floating-point data formats are provided. Subset implementations of the VAX architecture may not include all four data types. Operating system software may emulate omitted instructions and may utilize user-mode stack space during emulation. F__ floating and D__ floating are 4 and 8 bytes long, respectively. F__ floating data yields approximately 7 decimal digits of precision, while D__ floating yields approximately 16 decimal digits of precision. G__ floating is also 8 bytes long. Because of the different arrangement of the fraction and exponent parts, its precision is approximately 15 decimal digits. However, G__ floating has a wider range of exponents. H__ floating is 16 bytes long with a 15-bit exponent and 113-bit fraction. As a result, its precision is approximately 33 decimal digits.

Integer data is stored as binary values. An integer can be stored in a byte, word, longword, quadword, or in an octaword. A byte is 8 bits, a word is 2 bytes, a longword is 4 bytes, a quadword is 8 bytes, and an octaword is 16 bytes. The processor can interpret an integer as either a signed value (sign is determined by the high-order bit) or an unsigned value.

Numeric-string data is a representation of fixed quantities using 1 byte of the string for each decimal digit. The variety of external data arrangements demands a variety of matching numeric string forms; particularly, it is necessary to know whether the sign of the number appears in the first byte or as part of the last byte.

Packed decimal data is stored in a string of bytes. Each byte is divided in half forming two nibbles (4 bits = 1 nibble). One decimal digit is stored in each nibble. The first, or most significant digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.

Variable-length bit field data is small integers packed together in a larger data structure. Basically, they are used to increase memory efficiency.

Queue data is held in circular, doubly linked lists (that is, each entry is accompanied by two longwords—one longword tells the location of the succeeding entry, the other specifies the location of the preceding entry). Two kinds of queue data exist—absolute queues that use absolute addresses, and relative queues that use relative addresses.

The address of any data item is the address of the first byte in which the item resides. All integer, floating-point, packed-decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. It is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address. The native instruction set can interpret a bit field as a signed or unsigned integer.

▪ Stacks

A stack is an array of consecutively addressed data items referenced on a last-in/first-out (LIFO) basis using a general register. Data is added to and removed from the low address end of the stack. A stack grows toward lower addresses as items are added and shrinks toward higher addresses as items are removed.

A stack can be created anywhere in the program's address space and can use any register to point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software refers to its stack data structure, called the user stack, through a general register designated as the stack pointer (SP). When you run a program image, the operating system automatically provides the address of the area designated for the user stack.

A stack is an extremely powerful data structure because it can be used to pass arguments to routines. In particular, the stack structure enables the coding of reentrant routines because the processor can handle routine linkages automatically using the stack pointer. Routines can also be recursive: arguments can be saved on the stack for each successive call of the same routine.

▪ Condition Codes

Condition codes are used to indicate the type of result produced by an instruction. The codes are stored as bits in the processor status longword. Four conditions are coded—carry, negative, overflow, and zero.

-
- Carry condition code—indicates that the last instruction had a *carry* out of or a borrow from the most significant bit of the result.
-

-
- Negative condition code—indicates that an instruction produced a negative result.
-
- Overflow condition code—indicates that an instruction produced a result that could not be represented in an operand or that there was a conversion error.
-
- Zero condition code—indicates that the last instruction produced a zero result.
-

▪ Exceptions and Interrupts

The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software—exceptions and interrupts. Some exceptions, for example, arithmetic traps affect an individual process only. Other exceptions, for example, machine checks affect the system as a whole. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations called vectors to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the system control block base register, which the operating system loads with the physical address of the base of the system control block, where the exception and interrupt vectors are contained. The processor locates each vector by using a specific offset into the system control block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute.

To handle interrupt requests, the processor enters a special systemwide context. In the systemwide context, the processor executes in kernel mode using a special data structure called the interrupt stack. The interrupt stack cannot be referenced by any user-mode software because the processor selects the interrupt stack only after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the *return from exception or interrupt* instruction, the processor returns control to the previous level.

▪ Input/output Control

An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system's memory management software. That portion of physical address space in which device controller registers are located is called I/O space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller's registers as if they were physical memory locations. Software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. If interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver's interrupt service routine if it is not currently executing on a higher-priority interrupt level.

Chapter 3 ■ VAX Architecture Characteristics

The VAX architecture defines a *functional* behavior that is consistent throughout the family of VAX processors. From a programming point of view, the user environment is consistent. Characteristics of this consistency include sharing address space, sharing data, register usage, memory usage, restartability, interrupts and errors, and I/O structure.

■ Programming Environment

Within the context of any one process, user-level software controls its execution using the instruction sets, the general registers, and the processor status word. Within the multiprogramming environment, the operating system controls the system's execution using a set of special instructions, the processor status longword, and the processor registers.

Processor Status Longword

A processor register called the processor status longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the PSL are the processor status word (PSW) that is available to the user process. The high-order 16 bits provide privileged control of the system.

The PSL fields can be grouped by functions that control

-
- The access mode of the current instruction.
 - The instruction set in execution.
 - Interrupt processing.
-

Processor Access Modes

In a multiprogramming system, the processor must provide protection and sharing for the processes competing for system resources. The mechanism for protection in this system is the processor's access mode. The access mode is responsible for determining the

-
- Instruction execution privileges (which instructions the processor will execute).
 - Memory access privileges (which locations in memory the current instruction can access).
-

The processor executes code either in an interrupt context or in process context. In the interrupt context, all normal processing is halted until the interrupt is serviced. In the process context, the processor operates in one of four modes—kernel, executive, supervisor, and user. Kernel is the most privileged mode and user is the least privileged.

The processor executes in user mode in one process context or another. When a user process needs privileged services, it calls for those services. Then the processor executes the services either in the process's access mode or, temporarily under operating system control, in a more privileged mode. Only in kernel mode can the processor

-
- Execute an instruction that halts the processor.
-
- Load and save process context or access the internal processor registers controlling memory management.
-
- Access privileged processor registers.
-

The ability to execute code in a higher-privileged mode is controlled by the operating system. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in a less privileged mode.

Special Instructions

The VAX instruction set contains instructions that enable user-mode software to obtain privileged services without jeopardizing the integrity of the operating system. They are

-
- Change mode instructions (CHMK, CHME, CHMS, CHMU).
-
- PROBE instructions.
-
- Return from exception or interrupt (REI) instruction.
-

User-mode software can obtain privileged services with a standard *call* instruction. The operating system's service dispatcher issues an appropriate *change mode* instruction before actually entering the procedure. A *change mode* instruction is a special trap instruction similar to a service call instruction.

The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection against the privileges of the caller who requested access to a particular location. This makes the operating system provide services that execute in privileged modes to less privileged callers while preventing the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the REI instruction. The instruction is the only way to decrease the privilege of the processor's access mode. An REI instruction restores both the program counter and the processor state. This ensures the interrupted process's execution resumes at the point where it was interrupted.

When the operating system schedules a context switch, the context switching procedure uses the *save process context* (SVPCTX) and *load process context* (LDPCTX) instructions. The operating system's context switching procedure locates the hardware context to be loaded by updating a processor register.

Data Sharing

Data or instructions may be shared among various entities including programs, processors, and I/O devices. Entities sharing data may do so *explicitly* by referencing the same data or *implicitly* by referencing different items within the same physical memory location.

In VAX architecture, implicit sharing is transparent to the programmer. The memory system is implemented so the mechanism of access for independent modification is the byte. Not that this does not imply a maximum reference size of one byte but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, locations 0 and 1 contain the values 5 and 6, respectively. One process executes an INCB 0 instruction (increment by 1 the byte at location 0) and another executes an INCB 1 instruction. Regardless of the order of execution (including effectively simultaneous execution) the final contents must be 6 and 7.

Data Access Synchronization

Access to explicitly shared data that may be written must be synchronized. Before accessing shared writable data, the programmer must acquire control of the data structure. Seven instructions are provided to permit interlocked access to a control variable.

-
- The *branch on bit set and set, interlocked* (BBSSI) and *branch on bit clear and clear, interlocked* (BBCCI) instructions make a read and a subsequent write reference to a single bit within a single byte in an interlocked sequence.
-
- The *add aligned word, interlocked* (ADAWI) instruction makes a read and a subsequent write operation to a single aligned word in an interlocked sequence to allow counters to be maintained without other interlocks.
-

-
- The *insert at queue head, interlocked* (INSQHI), *insert at queue tail, interlocked* (INSQTI), *remove from queue head, interlocked* (REMQHI), and *remove from queue tail, interlocked* (REMQTI) instructions make a series of aligned longword reads and writes in an interlocked method to allow queues to be maintained without other interlocks.
-

Use of these instructions guarantees that no read operation within the synchronizing part of these instructions can occur between the synchronized reads and the writes of the instructions. Such instructions are implemented so that faults cannot cause the data structure to be locked for an extended period. On the processor, only interlocking instructions are locked out by the interlock.

▪ Registers

VAX processors contain three types of registers used during execution—general registers, processor registers, and input/output registers. The general registers are used as counters and pointers, and some are available for use by programmers. Processor registers perform system functions and normally are not used for other purposes. The input/output registers function in the control and status reporting of peripheral devices.

General Registers

VAX provides sixteen general registers for temporary storage of addresses and data. General registers are identified as R_n where n is an integer in the range 0 through 15. These registers do not have memory addresses. They are accessed either explicitly by specifying the register number in an instruction operand specifier, or implicitly by machine operations that automatically reference specific registers. Certain registers have specific uses and special names.

-
- Register R15—is the *program counter* (PC). The processor updates the register to address the next byte of the program. The PC is not used as a temporary, accumulator, or index register.
 - Register R14—is the *stack pointer* (SP). Several instructions make implicit references to the SP. Most software assumes that the SP points to memory set aside for use as a stack. There is no restriction on the explicit use of other registers (except PC) as stack pointers. Those instructions that make implicit references to the stack always use the SP.
-

-
- Register R13—is the *frame pointer* (FP). The VAX procedure call convention builds a data structure called a *stack frame* on the stack. The *call* instructions load the FP with the base address of the stack frame, and the *return* instruction depends on the FP containing the address of a stack frame. Further, VAX software depends on maintenance of the FP for correct reporting of certain exceptional conditions.
-
- Register R12—is the *argument pointer* (AP). The VAX procedure call convention uses a data structure called an *argument list*. The conventions need the AP as the base address of the argument list. *Call* instructions load the AP in accordance with that convention. There is no hardware or software restriction on the use of the AP for other purposes.
-
- Registers R6 through R11—these registers have no special significance to either the hardware or the operating system. Specific software assigns uses for each register.
-
- Registers R5 through R0—these registers are available for any use by software. But, they are also loaded with specific values by those instructions whose execution must be interruptible — the character string, decimal arithmetic, cyclic redundancy check, and polynomial instructions. The specific instruction descriptions identify which registers are used and what values are loaded into them.
-

The general philosophy of register allocation is high-numbered registers have the most global significance, low-numbered registers are used for the most temporary, local purposes. While there is no technical basis for this rule, it is a matter of convention followed by both hardware and system software. Thus, high-numbered registers are used for pointers needed by all software and hardware, and low-numbered registers are used for the working storage of string-type instructions. Similarly, the VAX procedure call software convention regards registers R0 and R1 as so temporary that they are not saved on calls. This is because R0 and R1 are used to return function values. Table 3-1 lists the use of general registers.

Table 3-1 • General Register Usage

Registers	Hardware Use	Conventional Software Use
PC (R15)	Program counter	Program counter
SP (R14)	Stack pointer	Stack pointer
FP (R13)	Frame pointer saved and loaded by CALL, used and restored by RET instruction	Frame pointer; condition signaling
AP (R12)	Argument pointer saved and loaded by CALL, restored by RET instruction	Argument pointer (base address of argument list)
R6:R11	None	Any
R3, R5	Address counter in character and decimal instructions	Any
R2, R4	Length counter in character and decimal instructions	Any
R1	Result of POLYD instruction; address counter in character and decimal instructions	Result of functions (not saved or restored on procedure call)
R0	Results of POLY, CRC instructions; length counter in character and decimal instructions	Results of functions, status of services (not saved or restored on procedure call)

A reference to the stack pointer (SP) can access one of five general stack pointers—executive, interrupt, kernel, supervisor, or user stack pointers depending on certain conditions. The conditions are the values of the *current mode* and *interrupt stack* bits in the processor status longword. Also, the *move to processor register* (MTPR) and *move from processor register* (MFPR) instructions can access those stack pointers including the currently selected stack pointer. This is also true for those processors whose executive, kernel, supervisor, and user stack pointers reside in the *process control block* PCB only.

Processor Registers

Processor registers reside in the processor register space. They are sometimes called *internal registers* or *privileged registers*. These registers perform control and status functions. They are accessible through the MTPR and MFPR instructions. These instructions can be invoked from the kernel mode only.

Context switching is the act of suspending an executing process and starting the execution of another. With the exception of the memory mapping and asynchronous system trap registers, processor registers are loaded from the processor control block (PCB) during a *context load* operation. During a *context save* operation, the registers are written to the PCB. In some VAX processors, *scratchpad* registers are used as an intermediate step in the read/write operation.

Depending on the model of processor, accessing processor registers using the MTPR and MTFR instructions may render invalid data. In some VAX processors, all or some of the processor registers reside in the PCB only. In those processors, the MTPR and MTFR instructions must be directed to the corresponding PCB location. VAX processors with processor registers in hardware scratchpads need not access the corresponding PCB locations.

▪ Clock Registers

There are two clocks in VAX processors: a time-of-year clock and an interval clock. The time-of-year clock register is used to measure the duration of power failures and is needed for unattended restart after a power failure. The interval clock is used for accounting, time-dependent events, and to maintain the date and time.

Time-of-year Clock The time-of-year clock is a longword register. It forms an unsigned 32-bit binary counter that is driven by a precision clock source. The clock operates at a minimum accuracy of 0.0025 percent. After approximately 497 days, the clock cycles to zero. As an option, the register can have an emergency power supply. The power supply may contain a battery that can operate for several hours. The register does not gain or lose time during the transition to or from the emergency power supply. The battery is usually automatically recharged.

Should the battery fail and the clocking is not accurate, the register is cleared after power is applied. Then the clock is started.

Interval Clock The interval clock provides an interrupt at programmed intervals. The counter is incremented at microsecond intervals with a minimum accuracy of 0.01 percent. The clock interface consists of three registers — the *interval count* register (ICR), the *next interval count* register (NICR), and the *interval clock control/status* register (ICSS).

The *interval count* register is a read-only register that is incremented every microsecond. It is automatically loaded from the *next interval count* register. If the interrupt is enabled, an interrupt is initiated when the *interval count* register is loaded.

The *next interval count* register is a write-only device that holds the value to be loaded into the *interval count* register when it overflows. The value is retained when the *interval count* register is loaded. The *next interval count* register is capable of being loaded regardless of the current values of the other two registers.

The *interval clock control/status* register contains control and status information for the interval clock. The register contains the following bits:

-
- *Run* bit—when the *run* bit is set, the interval count register increments each microsecond. When the bit is reset (0), the interval count register does not increment. During bootstrap procedures, the *run* bit is cleared.
-
- *Transfer* bit (XFR)—is a write-only bit. The *next interval count* register is transferred to the *interval count* register when this bit is set.
-
- *Signal* bit (SGL)—is a write-only bit. If the *run* bit is reset (0), the *interval count* register is incremented by one each time this bit is set.
-
- Software interrupt request (IE) bit—when this bit is set and the *interval count* register overflows, an interrupt request is generated. When this bit is reset, no interrupt is requested. If the hardware interrupt request bit is set and then this bit is set, an interrupt is requested.
-
- Hardware interrupt request (INT) bit—this bit is set by hardware whenever the *interrupt count* register overflows. If the IE bit is set and this bit is set, an interrupt is requested. If this bit is reset with a MTPR instruction, the *clock tick interrupt* is enabled.
-
- Error (ERR) bit—Whenever the *interval count* register overflows and the INT bit is set, this bit is set. This bit indicates that a *clock tick* was missed. This bit is cleared by a MTPR instruction.
-

NOTE

Subset processors may omit the *interval count* and *next interval count* registers. These processors are required to implement the software interrupt (IE) bit of the interval clock control/status register. If this bit is set, an interrupt request is generated every 10 milliseconds.

The interval clock is set by loading the negative of the desired interval into the NIC register. Then invoking an MTPR instruction enables interrupts, loads the *interval count* register with the interval stored in the *next interval count* register, and the *run* bit is set. Every *interval count* microsecond sets the INT bit and invokes an interrupt request. The interrupt routine should execute an MTPR instruction to clear the interrupt. If the INT bit is not reset by the next *interval count* register overflow, the *err* bit is set.

▪ Console Terminal Registers

The console terminal is accessed through four processor registers. Two registers are used for reception and two are used for transmission. There are control/status registers and data buffer registers for both reception and transmission. A status byte is used to determine the success or failure of a read or write operation. The status byte is transmitted to the operating system on completion of every *read*, *write*, or *read status* operation.

Receive Registers During bootstrap procedures, the console receive control/status register is initialized to zero. Whenever data is received, the *done* bit is set by the console. If the register's *interrupt enable* bit is set, an interrupt is requested. If the *done* bit is set and the software sets the *interrupt enable* bit, an interrupt is requested. If the data received contained an error, the *error* bit of the console receive data buffer register is set. The data received is stored in the data field of the register. When a MFPR instruction is executed, the *done* bit is reset. If the value in the ID field of the console receive data buffer register is zero, the data is from the console terminal. If the value of the ID field is other than zero, the entire register is implementation-dependent.

Transmit Registers During bootstrap procedures, the *console transmit control/status* register is initialized with all the bits reset except for the *ready* (RDY) bit which is set. Whenever the console transmitter is not busy, it sets the *ready* bit. And if the register's *interrupt enable* bit is set, an interrupt is requested. Also, if the *ready* bit is set, then the *interrupt enable* bit is set, an interrupt is requested. The software can send data by writing it to DATA. When an MTPR instruction is executed with the transmit data buffer as the destination operand, the *ready* bit is cleared. If the ID bit is zero, the data is sent to the console terminal. If the ID bit is set, the entire register is implementation-dependent.

▪ Process Control Block Base Register

The process control block base (PCBB) register is an internal privileged register containing the physical longword address of the process control block (PCB). The process control block contains the switchable process context. The context is collected into a compact form for ease of movement to and from the privileged registers.

In most operating systems, there is additional software context for each process. However, the following description is limited to the hardware process control block. See Figure 3-1 for an illustration of the process control block. Table 3-2 contains a description of the process control block longwords.

Longword		BIT	BIT
		3	0
		1	0
00	KERNEL MODE STACK POINTER		
01	EXECUTIVE MODE STACK POINTER		
02	SUPERVISOR MODE STACK POINTER		
03	USER MODE STACK POINTER		
04	REGISTER R0		
05	REGISTER R1		
06	REGISTER R2		
07	REGISTER R3		
08	REGISTER R4		
09	REGISTER R5		
10	REGISTER R6		
11	REGISTER R7		
12	REGISTER R8		
13	REGISTER R9		
14	REGISTER R10		
15	REGISTER R11		
16	ARGUMENT POINTER		
17	FRAME POINTER		
18	PROGRAM COUNTER		
19	PROCESSOR STATUS LONGWORD		
20	PROGRAM REGION BASE REGISTER		
21	PROGRAM REGION LENGTH REGISTER		
22	CONTROL REGION BASE REGISTER		
23	CONTROL REGION BASE REGISTER		
		3 2 2 2 2 2	0
		1 7 6 4 3 2 1	0

NOTES:

1. Asynchronous Trap Pending Field
2. Enable Performance Monitor Field

Figure 3-1 ■ Hardware Process Control Block

Table 3-2 ■ Process Control Block Definition

Long-word	Bits	Mnemonic	Description
0	31:0	KSP	Kernel Stack Pointer. Contains the stack pointer to be used when the value of the current access mode field in the processor status longword (PSL) is 0 and interrupt stack (IS) is 0.
1	31:0	ESP	Executive Stack Pointer. Contains the stack pointer to be used when the value of the current access mode field in the PSL is 1.
2	31:0	SSP	Supervisor Stack Pointer. Contains the stack pointer to be used when the value of the current access mode field in the PSL is 2.
3	31:0	USP	User Stack Pointer. Contains the stack pointer to be used when the value of the current access mode field in the PSL is 3.
4:17	31:0	R0:R11, AP, FP	General registers 0 through 11, argument pointer, and frame pointer.
18	31:0	PC	Program counter
19	31:0	PSL	Processor status longword
20	31:0	P0BR	Base register for the page table describing the process virtual addresses from 0 through 1,073,741,823 (decimal) ($2^{30} - 1$).
21	21:0	POLR	Length register for the page table located by base register P0. Describes the effective length of the page table.
	23:22	MBZ	Must be zero (0)
	26:24	ASTLVL	Contains the access mode number established by software of the most privileged access mode for which an asynchronous system trap is pending. Controls the triggering of the asynchronous system trap delivery interrupt during <i>return from exception</i> or <i>interrupt</i> instructions.

Table 3-2 ■ Process Control Block Definition (Cont.)

Long-word	Bits	Mnemonic	Description
			ASTLVL Field
			Asynchronous system trap pending for access mode:
			0 (kernel)
			1 (executive)
			2 (supervisor)
			3 (user)
			4 No traps pending
			5:7 Reserved
	31:27	MBZ	Must be zero (0)
22	31:0	P1BR	Base register for the page table. Describes the process virtual addresses from 1,073,741,824 through 2,147,483,647 (2^{30} through $2^{31} - 1$).
23	21:0	P1LR	Length register for the page table. Located by base register P1. Describes the effective length of the page table.
	30:22	MBZ	Must be zero (0)
	31	PME	Performance Monitor Enable. Controls a signal visible to an external hardware performance monitor. This bit is set to identify those processes for which monitoring is desired, and to permit their behavior to be observed without interference from other system activity

NOTE

Software symbols are defined for these locations by using the prefix PTX\$L and the mnemonics shown above. The prefix and mnemonic must be separated by an underscore character. For example, should you wish to specify the supervisor stack pointer register, the software symbol is: PTX\$L_ SSP. There are two exceptions to this symbology: longwords 21 and 23. The symbols for those words are:

PXT\$L_ POLRASTL (longword 21)

PTX\$L_ P1LRME (longword 23)

A process must be executing in one particular mode to alter its context switching fields. First the process stores the new value in the memory image of the process control block. Then it moves the value to the appropriate privileged register. This protocol is used because the process control block context switching fields are read-only fields. The context switching fields are POBR, P1BR, POLR, P1LR, ASTLVL, and PME.

NOTE

The ASTLVL and PME fields of the process control block are in registers when the process is executing. In order to access the fields, two privileged registers are provided. These are the AST Level register and the Performance Monitor Enable (PME) register.

▪ *System Identification Register*

The system identification (SID) register is a read-only device that specifies the processor type. The entire register is included in the error log and the type field may be used by software to distinguish processor types. The register is divided into two fields—*type* and *type specific*. The *type* field identifies the model of the processor. The *type specific* field varies among the models but contains specific identification for that model.

Input/Output Registers

Input/output registers are also known as *peripheral device control/status and data* registers. These registers are in the physical address space. They can be manipulated by memory reference instructions. Use of general instructions permits virtual address mapping and protection mechanisms to be used when referencing I/O registers. An area of the I/O physical address space maps through to the UNIBUS addresses. This area is called the UNIBUS space. I/O registers satisfy the following conditions:

-
- All registers must be aligned on natural boundaries.
-
- The physical address of an I/O register must always be an integral multiple of the register size in bytes (which must be a power of 2).
-
- References using a length attribute other than the length of the register and/or an unaligned reference may produce *unpredictable* results. For example, a byte reference to a wordlength register will not necessarily respond by supplying or modifying the byte addressed.
-

-
- In peripheral devices, error and status bits that may be asynchronously set by the device are usually cleared by software writing a one to these bits, and are not affected by writing a zero. This is to prevent resetting bits that may be asynchronously set between reading and writing a register.
-
- Only byte and word references of a read-modify-write type in UNIBUS I/O spaces are guaranteed to interlock correctly. References in the I/O space other than in UNIBUS spaces are undefined with respect to interlocking. This includes the BBSSI and BBCCI instructions.
-
- String, quadword, octaword, F__ floating, D__ floating, G__ floating, H__ floating, and field references in the I/O space result in *undefined* behavior.
-

▪ Stacks

Stacks are also called *pushdown lists* or *last-in/first-out (LIFO) queues*. They are an important feature of the architecture. They are used to perform various functions; for example, to

-
- Save the general registers including the program counter at entry to a sub-routine for restoration at exit.
-
- Save the program counter, processor status longword, and general registers at the time of interrupts and exceptions, and during context switches.
-
- Create storage space for temporary use or for nesting of recursive routines.
-

A stack is implemented in a VAX processor by a block of memory and a general register that addresses the *top* of the stack. The top of the stack is that location in the block containing the next candidate for removal. An item is added to the stack (*pushed on*) by decrementing the stack pointer register and storing the item at the address in the updated register. The pointer is decremented by the length of the item added to the stack to allow enough room for it. Conversely, the top item is removed (*popped off*) by adding the length of the item to the stack pointer after the last use of the item. These operations are built into the basic addressing mechanisms of VAX instructions. Thus any instruction can operate on the stack; it is seldom necessary to devote separate instructions to maintain the stack pointer.

There are two common programming errors associated with stacks: (1) adding more data than there is space to store, and (2) removing more data than was added. In order to catch those common programming errors, a stack is usually bound by inaccessible pages. By placing the stack in a block of memory between inaccessible pages, the programmer can be confident of finding such errors. The operating system initializes the stacks this way.

Many VAX processor operations make use of the stack implicitly; that is, without specifying the stack pointer in an operand. This occurs in instructions used in calling and returning from subroutines and in the processor sequences that initiate and terminate interrupt or exception service routines. In all such cases, the processor uses the stack addressed by R14.

This does not mean that exceptions, interrupts, and system services are performed on the same stacks employed by user-mode programs. The processor maintains five internal registers as pointers to separate blocks of memory to be used as stacks and uses one or another as a stack pointer depending on certain bits in the processor status longword. Whenever those bits change, the processor saves the stack pointer in a register selected by the old value of those bits. Then the processor loads the stack pointer from the register selected by the new value of these bits. There is one interrupt stack for the entire system. But the kernel, executive, supervisor, and user stacks are different for each process in the system. Figure 3-2 illustrates the relationships of the five stacks and multiple processes.

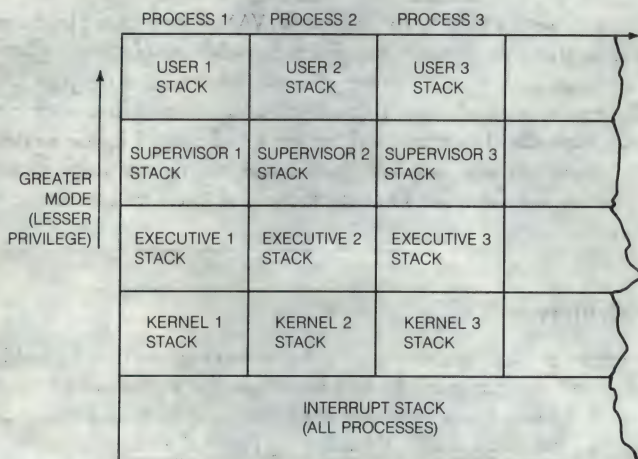


Figure 3-2 ■ Stacks by Mode versus Processes

This multiple-stack mechanism offers a number of advantages over a single-stack mechanism:

-
- User programs are not subject to sudden and nonreproducible changes in the data beyond the end of their stack. While it is bad practice to depend on such data, it would also be poor design to make it difficult to debug programs that did.
-
- The integrity of a privileged mode program cannot be compromised by a less privileged caller. Even if the caller has completely filled its own stack, the privileged code is in no danger of running out of space. Separate blocks of memory are allocated to the stack associated with each mode to prevent that situation.
-
- Privileged mode programs are not vulnerable to accidental or malicious destruction of the stack pointer by less privileged programs. Even if the user program uses the stack pointer as a floating-point accumulator, privileged code can use it as a stack pointer. To accomplish this, the processor saves the floating-point value and loads the pointer value when a mode change occurs.
-
- By allocating separate stacks for each mode, VAX processors can page most stack space dynamically while ensuring the availability of space for interrupt and page fault service. Interrupt service routines and the page fault handler may be invoked at any time and must have a small amount of stack available immediately without waiting for it to be paged in. Conversely, user programs may need very large stack spaces making it desirable to *swap out* those regions not in use. Only the kernel and interrupt stacks need be resident.
-

▪ Cache Memory

Cache memory or *cache* is a mechanism that reduces access time by making copies of recently used memory. In VAX family processors, the cache is implemented in such a way that it is transparent to software except for timing and error reporting/control. In cache, the following protocol is observed:

-
- Program writes to memory—followed by a peripheral output transfer—result in the output of the updated value.
-
- A peripheral input transfer—followed by a program reading of memory—results in a read of the input value.
-

-
- A write or modify operation—followed by a halt on one processor—followed by a read or modify operation on another processor—results in a read of the updated value. (Note that this applies only to customer-designed multiprocessor systems.)
-
- A write or modify operation—followed by a power failure — followed by restoration of power—followed by a read or modify operation—results in a read of the updated value. This occurs only if the duration of the power failure does not exceed the maximum nonvolatile period of the main memory or if the contents of memory were protected by an optional battery-operated emergency power supply.
-
- In multiprocessor systems, access to variables shared among processors can be interlocked by software executing interlocking instructions (ADAWI, BBCCI, BBSSI, INSQHI, INSQTI, REMQHI, or REMQTI). In particular, the write device must execute an interlocking instruction after the write to release the interlock and the read device must execute a successful matching interlock instruction before the read.
-
- Accesses to I/O registers are not loaded into the cache.
-

▪ Restartability

VAX architecture requires that all instructions be restartable after a fault or interrupt that terminated execution before the instruction was completed. Generally, this means that modified registers are restored to the value they had at the start of execution. For some complex or iterative instructions, intermediate results are stored in the general registers. In this case, memory may have been altered, but the former case requires that no operand be written unless the instruction can be completed. For most instructions with only a single modified or written operand, this implies special processing only when a multibyte operand spans a protection boundary. Spanning the boundary makes necessary the testing of the accessibility of both parts of the operand.

Instructions that store intermediate results in general registers do not compromise system integrity. They ensure that any addresses stored or used are virtual addresses subject to protection checking. Furthermore, they ensure that any state information stored or used does not result in a sequence that cannot be interrupted or terminated.

Instruction operands that are peripheral device registers having access side effects may produce *unpredictable* results due to the instruction restarting after faults (including page faults) or interrupts. To ensure no interrupts, programmers must avoid operand specifier addressing modes 9, 11, 13, and 15, and the indexed forms of these modes. (Refer to Chapter 5 for details of addressing modes.) However, the hardware may allow interrupts for these modes in order to minimize interrupt latency.

Memory modifications are produced as a by-product of instruction execution; for example, memory access statistics. These modifications are specifically excluded from the constraint that memory may not be altered until the instruction can be completed. Instructions that abort are constrained only to ensure memory protection; for example, the registers can be changed.

■ Interrupts and Errors

Underlying the VAX architectural concept of an interrupt is the notion that an interrupt request is a static condition—not a transient event—and can be sampled by a processor at appropriate times. Further, if an interrupt is no longer needed before a processor has honored that interrupt request, the interrupt request can be removed without consequence. Any instruction that changes the processor's interrupt priority level to enable a pending interrupt allows the interrupt to occur before executing the next instruction. Similarly, if processor priority permits, instructions that generate requests at the software interrupt levels allow the interrupt to occur before executing the next instruction.

Processor errors that are consistent with instruction completion create high-priority interrupt requests. Otherwise, they terminate instruction execution with an exception. Error notification interrupts may be delayed from the apparent completion of the executing instruction at the time of the error. But, if enabled, the interrupt is requested before processor context is switched.

Chapter 4 ■ Data Representation

VAX instructions use a variety of types of data. The data must be presented in a form that is acceptable to the instructions. The acceptable forms are described in the ensuing paragraphs. They are

-
- Character string data
 - Floating-point data
 - Integer data
 - Numeric string data
 - Packed decimal data
 - Queue data
 - Variable length bit field data
-

NOTE

In the following discussions of floating-point and integer data, the address of the data in memory is the address of the byte of the data with the lowest address. In illustrations, this lowest byte is shown on the right (bits 0 through 7). In text, when the word *right* is used to describe the position of a byte, the lowest byte is the byte being discussed.

▪ Character String Data

Character strings are used to represent names, data records, or text. The instructions allow you to copy, search, concatenate, and translate strings. A character string is a contiguous sequence of bytes in memory and is specified by two attributes—the address (A) of the first byte of the string, and the length (L) of the string in bytes. The length of a string is in the range 0 through 65,535. A string with a length of 0 is called a *null string*. Null strings have no bytes. No memory is referenced; hence, the address need not be valid. The format of a character string is illustrated in Figure 4-1. The address of a string specifies the first character of a string as shown in Figure 4-2.

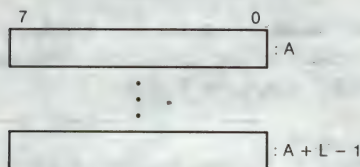


Figure 4-1 ■ Character String Format

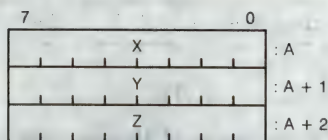


Figure 4-2 ■ Character String Address

■ Floating-point Data

The VAX instruction set supports floating-point data in longwords, quadwords, and octawords. Four types of floating-point data are available. Two types are eight bytes long (D__ floating and G__ floating), one type is four bytes long (F__ floating), and the last is sixteen bytes long (H__ floating).

NOTE

An exponent value of zero with a sign bit that is zero is taken as *reserved*.

D__ floating Data

D__ floating data is sometimes called *double floating* or *double-precision floating*. It is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right starting with 0 and terminating with 63 as shown in Figure 4-3.

D__ floating data is specified by the address of the byte containing the first bit. The form of D__ floating data is identical to F__ floating data except for an additional 32 low-significance fraction bits. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 6. This is illustrated by the widening arrow in Figure 4-3. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 with a sign bit of 0 indicates the data has a value of zero. Exponent values of 1 through 255 indicates true binary exponents of -127 through $+127$. Floating-point instructions processing a reserved operand take a reserved operand fault. The values of D__ floating data are in the approximate range 0.29×10^{-38} through 1.7×10^{38} . The precision is approximately one part in 2^{25} or 16 decimal digits.

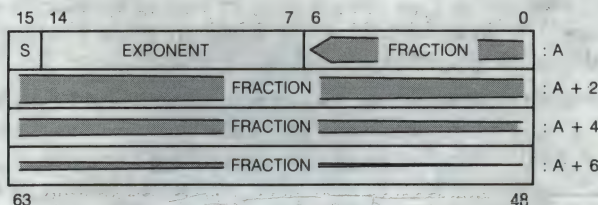


Figure 4-3 • D__ floating Data Format

F__ floating Data

F__ floating data is sometimes called *floating* or *single-precision floating* and is four contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right starting at 0 and terminating with 31 as shown in Figure 4-4.

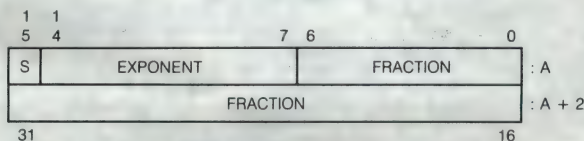


Figure 4-4 • F__ floating Data Format

F__ floating data is specified by the address of the byte containing the first bit. The form of F__ floating data is sign magnitude with bit 15 as the sign bit, bits 7 through 14 express an *excess 128* binary exponent, and bits 0 through 6 and 16 through 31 are a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0 with a sign bit of 0 indicates the data has a value of zero. Exponent values of 1 through 255 indicates true binary exponents of -127 through $+127$. Floating point instructions processing a reserved operand take a reserved operand fault. The values of F__ floating data are in the approximate range 0.29×10^{-38} through 1.7×10^{38} . The precision is approximately one part in 2^{23} , or approximately seven decimal digits.

G__ floating Data

G__ floating data is eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right starting with 0 and ending with 63 as shown in Figure 4-5.

The form of G__ floating data is sign magnitude with bit 15 as the sign bit, bits 4 through 14 expressing an *excess 1024* binary exponent, and bits 0 through 3 and 16 through 63 expressing a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0 with a sign bit of 0 indicates the data has a value of zero. Exponent values of 1 through 2047 indicate true binary exponents of -1023 through $+1023$. Floating-point instructions processing a reserved operand take a reserved operand fault.

The value of G__ floating data is in the appropriate range of 0.56×10^{-308} through 0.9×10^{308} . The precision is approximately one part in 2^{52} or fifteen decimal digits.

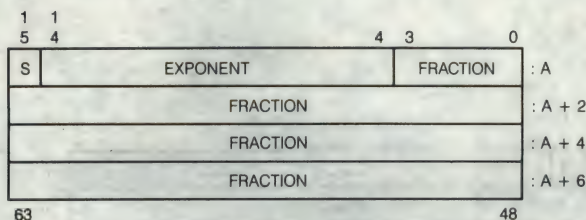


Figure 4-5 ■ G__ floating Data Format

H__ floating Data

H__ floating data is sixteen contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right starting with 0 and ending with 127 as shown in Figure 4-6. H__ floating data is specified by the address of the byte containing the first bit.

The form of H__ floating data is sign magnitude with bit 15 as the sign bit, bits 0 through 14 express an *excess 16384* binary exponent, and bits 16 through 127 express a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits increase in significance from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31. The 15-bit exponent field encodes the values 0 through 32,767. An exponent value of 0 with a sign bit of 0 indicates that the data has a value of zero. Exponent values of 1 through 32,767 indicate true binary exponents of $-16,383$ through $+16,383$. Floating-point instructions processing a reserved operand take a reserved operand fault.

The value of H__ floating data is in the approximate range 0.84×10^{-4932} through 0.59×10^{4932} . The precision is approximately one part in 2^{112} or thirty-three decimal digits.

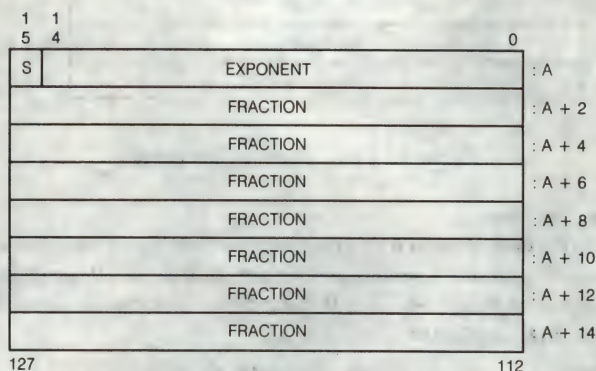


Figure 4-6 ■ H floating Data Format

■ Integer Data

VAX systems support integer data in 8-bit *bytes*, 16-bit *words*, 32-bit *long-words*, 64-bit *quadwords*, and 128-bit *octawords*. Integer data is stored in a binary format that can be signed or unsigned. As unsigned quantities, integers increment from 0. As signed quantities, the integers are represented in two's complement form. This means that positive numbers have a zero for the *most significant bit* (MSB); and the representation of a negative number is one greater than the bit-by-bit complement of its positive counterpart. Thus the MSB is always zero for positive values and one for negative values.

Byte Data

A byte is eight contiguous bits starting on an addressable byte boundary. The bits are numbered from the right starting with 0 as shown in Figure 4-7. The byte is specified by its address. When interpreted arithmetically, a byte is a two's complement integer with bits increasing in significance from 0 through 6 and with bit 7 designating the sign. The value of the integer is in the range of -128 through 127. For addition, subtraction, or comparison, VAX instructions provide direct support for interpreting a byte as an unsigned integer with bits of increasing significance starting at bit 0 and increasing to bit 7. The value of the unsigned integer is in the range of 0 through 255.



Figure 4-7 ■ Byte Data Format

Word Data

A word is two contiguous bytes and starts on an arbitrary byte boundary. The bits are numbered from the right starting with 0 as shown in Figure 4-8. Words are specified by their address which is the address of the byte containing the first bit. When interpreted arithmetically, a word is a two's complement integer with bits increasing in significance from 0 through 14, and bit 15 designating the sign. The value of the integer is in the range -32,768 through 32,767. For addition, subtraction, and comparison, VAX instructions provide direct support for interpreting a word as an unsigned integer with bits increasing in significance from bit 0 to bit 15. The value of an unsigned integer is in the range of 0 through 65,535.



Figure 4-8 ■ Word Data Format

Longword Data

A longword is four contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right starting with 0 as shown in Figure 4-9. A longword is specified by its address that is the address of the byte containing the first bit. When interpreted arithmetically, a longword is a two's complement integer with bits increasing in significance from 0 through 30 and with bit 31 designating the sign.

The value of the integer is in the range $-2,147,483,648$ through $2,147,483,647$. For addition, subtraction, and comparison, VAX instructions provide direct support for interpreting longwords as unsigned integers with bits increasing in significance from bit 0 to bit 31. The value of the unsigned integer is in the range of 0 through $4,294,967,295$.



Figure 4-9 • Longword Data Format

Quadword Data

A quadword is eight contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right starting with 0 and terminating with 63 as shown in Figure 4-10. A quadword is specified by its address that is the address of the byte containing the first bit. When interpreted arithmetically, a quadword is a two's complement integer with bits increasing in significance from 0 through 62, and bit 63 is the sign bit. The value of the integer is in the range -2^{63} to $(2^{63})-1$. The quadword data type is not fully supported by VAX instructions.

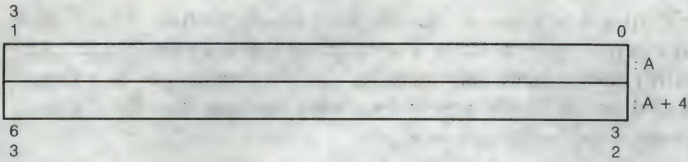


Figure 4-10 ■ Quadword Data Format

Octaword Data

An octaword is sixteen contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right starting with 0 and terminating with 127 as shown in Figure 4-11. Octawords are specified by the address of the byte containing the first bit. When interpreted arithmetically, an octaword is a two's complement integer with bits of increasing significance starting at bit 0 and terminating at bit 126, and bit 127 is the sign bit. The value of the integer is in the range -2^{127} to $(2^{127})-1$. The octaword data type is not yet fully supported by VAX instructions.

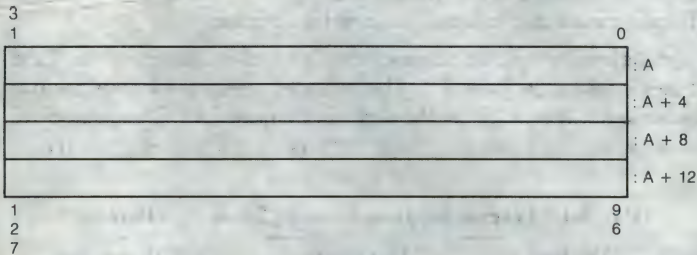


Figure 4-11 ■ Octaword Data Format

▪ Numeric String Data

Numeric string data is used to represent fixed scaled quantities in forms close to their external representations. For programs that are input/output intensive, rather than computation intensive, this presentation can be efficient. The decimal string form also provides greater precision than floating point and greater range than integer data types.

There are two forms of decimal data on VAX systems— numeric and packed. In numeric string data, each digit occupies one byte. In packed decimal strings, two digits are packed into one byte. Because the numeric string exactly represents many external data arrangements, it appears in several forms.

There are two forms of signed numeric strings. The first is called the *leading separate numeric string*; the second is called the *trailing numeric string*. In the leading separate numeric string, the sign appears before the first digit. In the trailing numeric string, the sign is superimposed on the last digit.

Leading Separate Numeric String Data

A leading separate numeric string is a contiguous sequence of bytes in memory. It is specified by two attributes—an address and a length. The address is the address of the first byte or the sign character. The length is the *length of the string in digits—not the length of the string in bytes*. The number of bytes in a leading separate numeric string is the length plus one. The address of the string specifies the byte of the string containing the sign. Digits of decreasing significance are assigned to bytes of increasing addresses.

The sign of a leading separate numeric string is stored in a separate byte. Valid sign bytes are listed in Table 4-1. The preferred representation for positive strings is the ASCII code 2B for the plus sign character. All subsequent bytes contain an ASCII digit character. Table 4-2 lists the ASCII digit characters.

Table 4-1 • Leading Separate Numeric String Sign Bytes

Sign	Decimal	Hexadecimal	ASCII character
+	43	2B	+
+	32	20	< blank >
-	45	2D	-

Table 4-2 ■ ASCII Digit Characters

Digit	Decimal	Hexadecimal	ASCII character
0	48	30	0
1	49	31	1
2	50	32	2
3	51	33	3
4	52	34	4
5	53	35	5
6	54	36	6
7	55	37	7
8	56	38	8
9	57	39	9

The length of a leading separate numeric string must be within the range of 0 to 31 (0 to 31 digits). The value of a zero length string is zero. It contains the sign byte only. Figures 4-12 and 4-13 show how to represent +123 and -123 in leading separate numeric string format.

7	4	3	0	
2		B		: A
3		1		: A + 1
3		2		: A + 2
3		3		: A + 3

Figure 4-12 ■ Positive Leading Separate Numeric String Format

7	4	3	0	
2		D		: A
3		1		: A + 1
3		2		: A + 2
3		3		: A + 3

Figure 4-13 ■ Negative Leading Separate Numeric String Format

Trailing Numeric String Data

A trailing numeric string is a contiguous sequence of bytes in memory. The string is specified by two attributes—an address and the length of the string. The address of the first byte of the string is the most significant digit. The length is the length of the string in bytes. Note that the address of the string specifies the byte of the string containing the most significant digit. Digits of decreasing significance are assigned to increasing addresses. All bytes of a trailing numeric string except the least significant digit byte must contain ASCII decimal (0 through 9) characters. See Table 4-2 for a list of the ASCII characters.

The highest addressed byte of a trailing numeric string represents an encoding of both the least significant digit and the sign of the numeric string. The numeric string instructions support any encoding. There are three preferred encodings used by VAX software

-
- overpunched numeric
-
- *unsigned numeric* in which there is no sign and the least significant digit contains an ASCII decimal digit character
-
- zoned numeric
-

Several variations in overpunched format have evolved because that format has been used for many years, and because various card encodings are used. These alternate forms are accepted on input. The normal form is generated on output of all operations. The valid representations of the digit and sign in each of the latter two formats is shown in Table 4-3.

Table 4-3 • Representation of Least Significant Digit and Sign

Digit	Decimal	Hexadecimal	ASCII Character	
			Normal	Alternate
Overpunch Format				
0	123	7B	{	*
1	65	41	A	1
2	66	42	B	2
3	67	43	C	3
4	68	44	D	4
5	69	45	E	5
6	70	46	F	6
7	71	47	G	7
8	72	48	H	8
9	73	49	I	9
-0	125	7D	}	†
-1	74	4A	J	None
-2	75	4B	K	None
-3	76	4C	L	None
-4	77	4D	M	None
-5	78	4E	N	None
-6	79	4F	O	None
-7	80	50	P	None
-8	81	51	Q	None
-9	82	52	R	None

* There are three alternate characters for this code: the zero (0), the left square bracket ([), and the question mark (?).

† There are three alternate characters for this code: the right square bracket (]), the colon (:), and the exclamation point (!).

Table 4-3 ■ Representation of Least Significant Digit and Sign (Cont.)

Digit	Decimal	Hexadecimal	ASCII Character	
			Normal	Alternate
Zoned Numeric Format				
0	48	30	0	None
1	49	31	1	None
2	50	32	2	None
3	51	33	3	None
4	52	34	4	None
5	53	35	5	None
6	54	36	6	None
7	55	37	7	None
8	56	38	8	None
9	57	39	9	None
-0	112	70	p	None
-1	113	71	q	None
-2	114	72	r	None
-3	115	73	s	None
-4	116	74	t	None
-5	117	75	u	None
-6	118	76	v	None
-7	119	77	w	None
-8	120	78	x	None
-9	121	79	y	None

The length of a trailing numeric string must be within the range of 0 to 31 (0 to 31 digits). The value of a zero length string is zero. It contains no bytes and no memory is referenced; hence the address need not be valid. Figures 4-14 and 4-15 show how to represent the value 123 in both positive and negative trailing numeric string format.

ZONED FORMAT OR UNSIGNED

7	4	3	0
3	1	: A	
3	2	: A + 1	
3	3	: A + 2	

OVERPUNCH FORMAT

7	4	3	0
3	1	: A	
3	2	: A + 1	
4	3	: A + 2	

Figure 4-14 ■ Positive Trailing Numeric String Format

ZONED FORMAT

7	4	3	0
3	1	: A	
3	2	: A + 1	
7	3	: A + 2	

OVERPUNCH FORMAT

7	4	3	0
3	1	: A	
3	2	: A + 1	
4	C	: A + 2	

Figure 4-15 ■ Negative Trailing Numeric String Format

▪ Packed Decimal String Data

A packed decimal string is a contiguous sequence of bytes in memory. The address and length specify a packed decimal string. The length is the *number of digits in the string—not the number of bytes*. Every byte of a packed decimal string is divided into two 4-bit fields called *nibbles*. Each nibble must contain decimal digits except the low nibble of the last byte that must contain a sign. The representation for the digits and sign is listed in Table 4-4.

Table 4-4 • Packed Decimal String Digits and Signs

Character	Decimal	Hexadecimal
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
+	*	†
-	‡	§

* This value can be 10, 12, 14, or 15 (hexadecimal).

† This value can be A, C, E, or F (hexadecimal).

‡ This value can be 11 or 13 (hexadecimal).

§ This value can be B or D (hexadecimal).

The preferred sign representation is 12 for a plus sign and 13 for a minus sign. The length is the number of digits in the packed decimal string (not counting the sign) and must be within the range of 0 through 31. If the number of digits is odd, the digits and the sign fit into $\text{length}/2 + 1$ bytes. When the number of digits is even, an extra 0 digit must appear in the high nibble (bits 4 through 7) of the first byte. The length in bytes of a string with an even number of bytes is $\text{length}/2 + 1$ bytes. The length is the integer portion only. The value of a zero-length packed decimal string is zero. It contains only the sign byte that also includes the extra 0 digit.

The address of the string specifies the byte of the string containing the most significant digit in its high nibble. Digits of decreasing significance are assigned to increasing byte addresses and from high nibble to low nibble within a byte. In Figure 4-16, + 123 (length 3) is represented in packed decimal format. In Figure 4-17, - 12 (length 2) is represented in packed decimal format.

7	4	3	0	
1	2			: A
3	12			: A + 1

Figure 4-16 ■ Positive Packed Decimal String Format

7	4	3	0	
0	1			: A
2	13			: A + 1

Figure 4-17 ■ Negative Packed Decimal String Format

▪ Queue Data

A queue is a list whose entries are specified by their addresses. Each queue entry is linked to the next by way of a pair of longwords. The first longword is the *forward link*. It specifies the location of the succeeding entry. The second longword is the *backward link*. It specifies the location of the preceding entry. VAX systems support two types of links—absolute and self-relative. Queues are named after the type of link used in the queue.

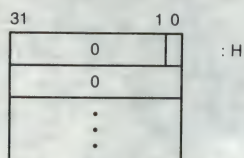
An absolute queue uses a link that contains the absolute address of the entry to which it points. A self-relative queue uses a link that contains a displacement from the present queue entry.

Queues require a header that is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry called the *head of the queue*. The backward link of the header is the address of the entry called the *tail of the queue*. Logically, the forward link of the tail points to the header.

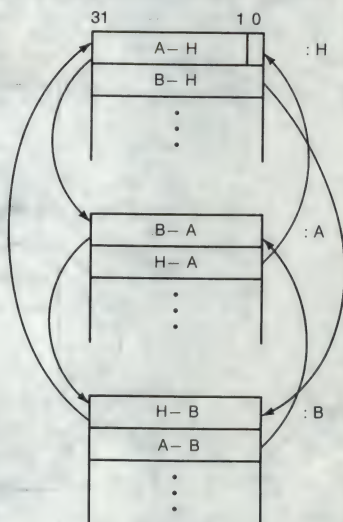
Self-relative queues are intended for use in situations where they are addressed by two separate processes. Each process may view the queues as residing in two separate locations in their respective virtual address spaces. The instructions that operate on self-relative queues are interlocked. When interlocked instructions only are used on the queue, the processes may be in separate machines with each process directly addressing the queue.

Absolute queues are somewhat simpler in structure than self-relative queues in that their pointers are virtual addresses. Also, the instructions that operate on these queues are not interlocked. In general, operations on absolute queues are somewhat faster than those on self-relative queues. However, absolute queues cannot be used when more than one processor is to access them. Also, the queues can be shared by two processes in the same processor only when both processes address the queue in the same section of their virtual address space. Figure 4-18 illustrates the format of the self-relative queue, and Figure 4-19 illustrates the format of the absolute queue.

EMPTY SELF-RELATIVE QUEUE (HEADER ONLY)



SELF-RELATIVE QUEUE WITH ONE ENTRY



SELF-RELATIVE QUEUE WITH TWO ENTRIES

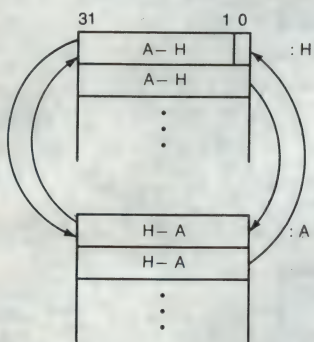
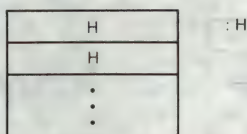
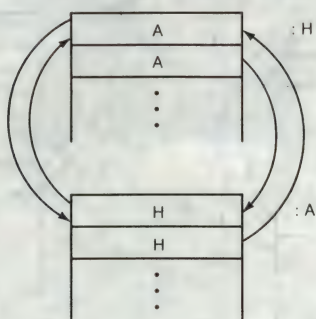


Figure 4-18 ■ Self-relative Queues

EMPTY ABSOLUTE QUEUE (HEADER ONLY—SIMPLE ENTRY ONLY)



ABSOLUTE QUEUE WITH HEADER AND OTHER ENTRY



ABSOLUTE QUEUE WITH HEADER AND TWO OTHER ENTRIES

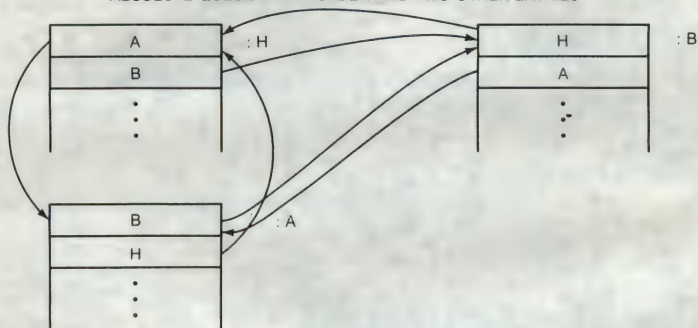


Figure 4-19 • Absolute Queues

▪ Variable Length Bit Field Data

The variable length bit field is a type of data used to store small integers packed together in a larger data structure. This conserves memory when many small integers are part of a larger structure. A specific case of the variable bit field is that of one bit. This form is used to store and access individual flags efficiently.

A variable bit field is from 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries and specified by three attributes—a base address, a bit position, and size.

The base address (A) is the address of a particular byte in memory chosen as a reference point for locating the bit field F. The bit position (P) is the signed longword specifying the bit displacement of the least significant bit of the field with respect to bit zero of the byte at address A. The size (S) is the byte integer length of field F expressed as a number of bits. Size must be between 0 and 32 bits inclusive. Figure 4-20 illustrates the variable length bit field where the field is the shaded area.

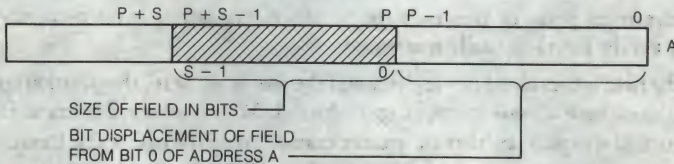


Figure 4-20 ▪ Variable Length Bit Field

For bit strings in memory, the position in bits can be either a positive or negative displacement within the range of -2^{31} through $(2^{31}) - 1$. It can be viewed as a signed 29-bit byte offset and a 3-bit bit-within-byte field as shown in Figure 4-21.

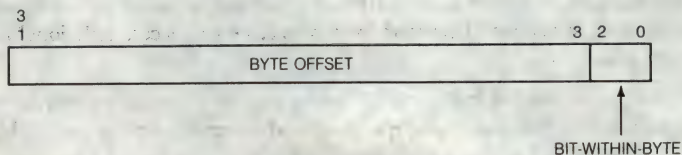


Figure 4-21 ■ Variable Length Bit Field in Memory

The sign-extended 29-bit byte offset is added to the address and the resulting address specifies the byte in which the field begins. The 3-bit bit-within-byte field encodes the starting position (0 through 7) of the field within that byte. VAX instructions provide direct support for the interpretation of a field as a signed or unsigned integer. When interpreted as a signed integer, it is the two's complement with bits increasing in significance from 0 through S-2 where bit S-1 is the sign bit. When interpreted as an unsigned integer, bits increase in significance from 0 through S-1. A field size of zero has a value of zero.

A variable bit field may be contained in zero to five bytes. From a memory management point of view only the minimum number of bytes necessary to contain the field is actually referenced.

If the field is contained in a register and the size is not zero, the position operand must have a value in the range 0 through 31 or a reserved operand fault occurs. If size plus position are greater than 32, then the operand is located in the concatenation of register $[n+1]$ and by register $[n]$ (that is $R[n+1] \text{ } \text{RR}[n]$). See Figure 4-22. The most significant bit of the specified field lies in $R[n+1]$ and the least significant bit of the specified field is located in $R[n]$.

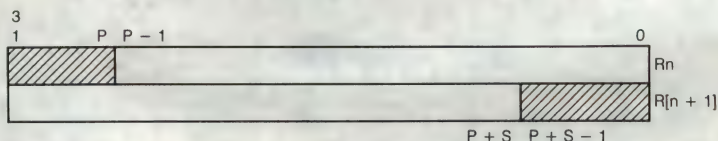


Figure 4-22 ■ Variable Length Bit Field in Register

To illustrate the variable length bit field with a positive displacement, assume the following variable length bit field attributes—base address (A) = B2204C01, position (P) = 29, and size (S) = 2. See Figure 4-23. The starting position of the field is bit 29; that is, the first bit of F is the twenty-ninth bit after bit zero of A as shown in Figure 4-23. Now that the starting bit position of field has been located, determine its length. To determine its length, apply the size attribute as shown in Figure 4-24.

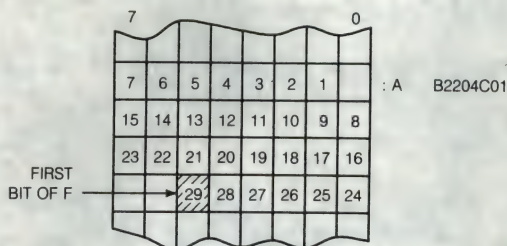


Figure 4-23 ■ Positive Displacement Variable Bit Field

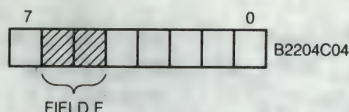


Figure 4-24 ■ Determining Length of Positive Displacement Field

To determine the length of negative displacement variable length bit field, assume the following attributes—base address (A) = 801134E3, position (P) = -7, and size (S) = 6. See Figure 4-25. The starting position of F is the seventh bit preceding the zero bit of address 801134E3 as shown in Figure 4-25. To determine the field length, apply the size attribute as in the previous example counting from lower to higher addresses as shown in Figure 4-26.

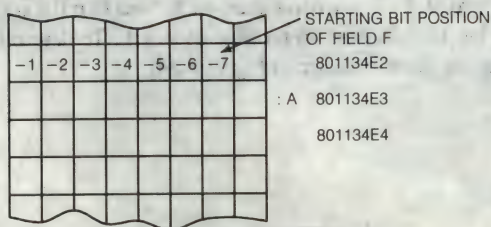


Figure 4-25 ■ Negative Displacement Variable Bit Field



Figure 4-26 ■ Determining Length of Negative Displacement Field

■ Data in Registers

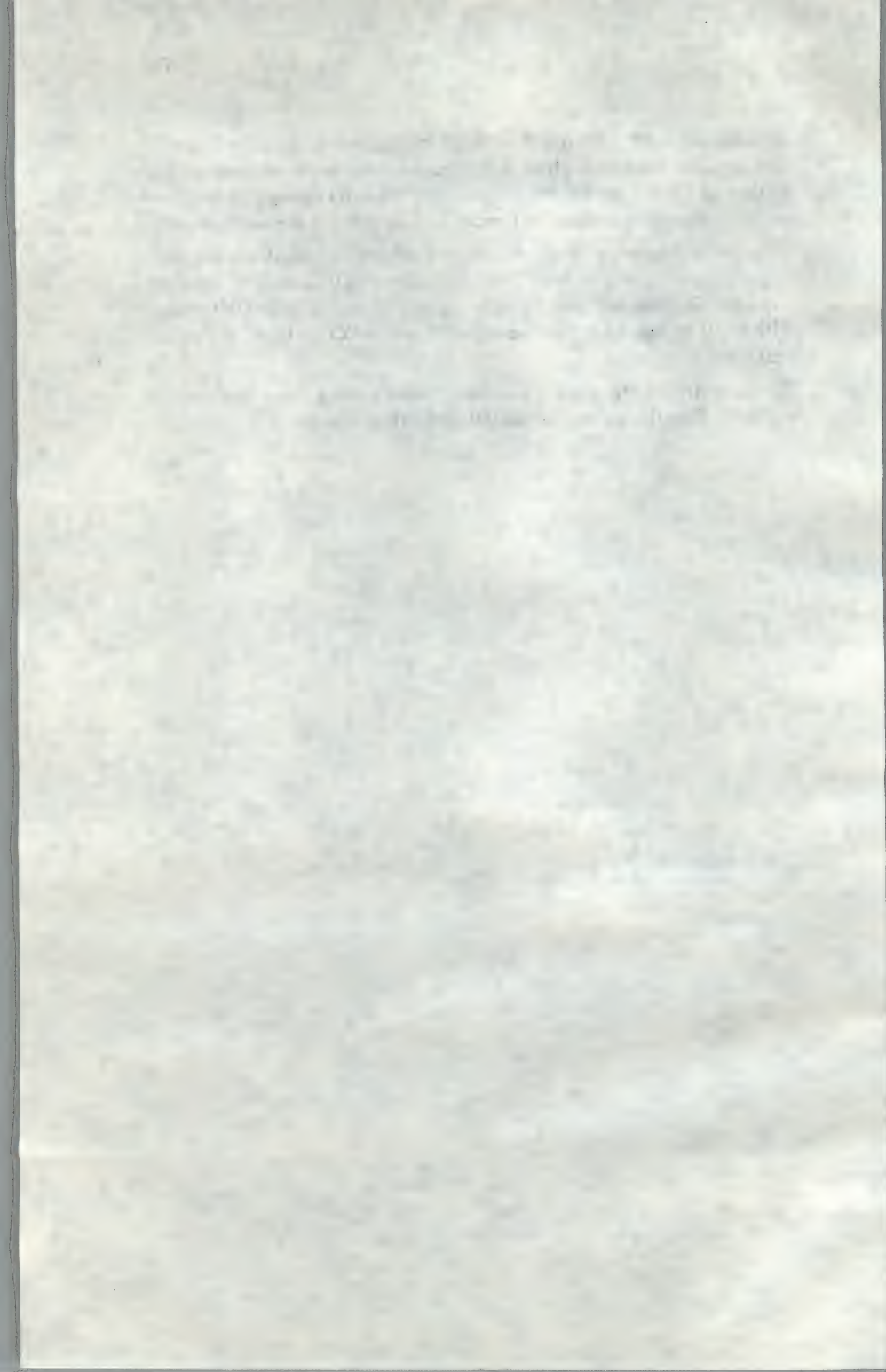
When byte, word, longword, or floating type data is stored in a register, the bit numbering in the register corresponds to the numbering in memory. A byte is stored in a register in bits 0 through 7. A word is stored in bits 0 through 15. Longword and F__ floating data is stored in register bits 0 through 31. A byte or word written to a register writes only bits 7 through 0 and 15 through 0 respectively. The higher bits are unaffected. A byte or word read from a register reads only bits 0 through 7 and 0 through 15, respectively. The other bits are ignored.

When quadword or D__ floating data is stored in a register, the data is stored in two adjacent registers. Because of program counter restrictions, wrap-around from register PC to register R0 is *unpredictable*. Bits 0 through 31 of the quadword or D__ floating data are stored in the first register. Bits 32 through 63 of the quadword or D__ floating data are stored in the second register.

An octaword or H__ floating data stored in a register is stored in four adjacent registers. Bits 0 through 31 of the data are stored in the first register, bits 32 through 63 are stored in the second register, bits 64 through 95 are stored in the third register, and bits 96 through 127 are stored in the fourth register.

With one restriction, a variable length bit field may be specified in the registers. The starting bit position (P) must be in the range 0 through 31. For quadword and D__ floating data, a pair of registers is treated as a 64-bit register with bits 0 through 31 in the base register and bits 32 through 63 in the adjacent register.

The VAX string instructions are unable to process string data types stored in registers. Thus there is no representation of strings in registers.



Chapter 5 ■ The Instruction Characteristics

The notation conventions, source statement format, and register addressing modes are described in this chapter. One must understand the notation conventions before being able to read and comprehend the instructions. Then the addressing modes can be studied. Addressing modes are related to the instruction format because the form of the instruction implicitly specifies the register addressing mode.

■ Notation Conventions

The notation conventions described here are for the assembler and instruction set only and do not apply to other syntax. The conventions cover the assembler, instruction operand, instruction operation, and range and extent notation.

Assembler Notation

The radix of the assembler is decimal. To express a hexadecimal number in assembler notation, the number must be preceded by a *caret* (`&`) and an uppercase X. For those keyboards without a caret character, the *up arrow* (`↑`) character is used. For example, in the instruction `MOVW #3456, -(SP)`, the assembler interprets the number 3456 as a decimal number. If 3456 is to be interpreted as a hexadecimal number, it must be preceded by a caret or up arrow and an uppercase X (`#↑X3456`).

Operand Notation

Operands are specified in the following way:

`<name> . <access__ type> <data__ type>`

where `<name>` is typically a mnemonic for the operand of the instruction.

The `<access__ type>` is a letter denoting the operand access type.

- a means to calculate the effective address of the specified operand. Address is returned in a longword that is the instruction operand. Context of address calculation is given by `<data__ type>`.
- b means there is no operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by `<data__ type>`.

- m means the operand is read, sometimes modified, and written. Note that this is *not* an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.
- r means the operand is read only.
- v means to calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword that is the instruction operand. Context of address calculation is given by `<data__ type>`. If the effective address is Rn , then the operand actually appears in $R[n]$, or in $R[n + 1]R[n]$.
- w means the operand is written only.

The `<data__ type>` is a letter denoting the data type of the operand.

- b denotes byte data
- d denotes D__ floating data
- f denotes F__ floating data
- g denotes G__ floating data
- h denotes H__ floating data
- l denotes longword data
- o denotes octaword data
- q denotes quadword data
- w denotes word data
- x denotes the first data type specified by instruction
- y denotes the second data type specified by instruction

Operation Notation

The operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to define the syntax formally. The syntax is assumed to be familiar to the reader and is summarized in Table 5-1.

Table 5-1 ■ Operation Notation Conventions

Notation	Meaning
+	addition
-	subtraction
×	multiplication
/	division (quotient only)
**	exponentiation
,	concatenation
←	is replaced by
=	is defined as
Rn or R[n]	contents of register Rn
PC	the contents of register R15
SP	the contents of register R14
FP	the contents of register R13
AC	the contents of register R12
PSW	the contents of the Processor Status Word
PSL	the contents of the Processor Status Longword
(x)	contents of memory whose address is x
(x) +	contents of memory whose address is x; x is incremented by size of operand referenced at x
-(x)	x decremented by size of operand to be referenced at x; contents of memory whose address is x
x:y	a modifier which delimits an extent from bit position x to bit position y inclusive
x1,x2,...,xn	a modifier that enumerates bits x1,x2, ... ,xn
x...y	x through y inclusive
{ }	braces used to indicate precedence
AND	logical AND
OR	logical OR
XOR	logical XOR

Table 5-1 • Operation Notation Conventions (Cont.)

Notation	Meaning
NOT	logical (1's) complement
LSS	less than signed
LSSU	less than unsigned
LEQ	less than or equal signed
LEQU	less than or equal unsigned
EQL	equal signed
EQLU	equal unsigned
NEQ	not equal signed
NEQU	not equal unsigned
GEQ	greater than or equal signed
GEQU	greater than or equal unsigned
GTR	greater than signed
GTRU	greater than unsigned
SEXT (x)	x is signed-extended to size of operand needed
ZEXT (x)	x is zero-extended to size of operand needed
REM (x, y)	remainder of x divided by y, such that x/y and REM (x,y) have the same sign
MINU (x, y)	minimum unsigned of x and y
MAXU (x, y)	maximum unsigned of x and y

The following conventions are used:

- Other than that caused by (x) + , or - (x), and the advancement of the program counter, only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed other than that replacement has the lowest precedence. Precedence is indicated explicitly by braces.
- All arithmetic, logical, and relational operators are defined in the context of their operand. For example, a plus sign (+) applied to floating operands means a floating add while the same sign applied to byte operands means an integer byte add. Similarly, LSS is a floating comparison when applied to floating operands; and LSS is an integer byte comparison when applied to byte operands.

-
- Instruction operands are evaluated according to the operand specifier conventions. The order in which operands appear in the instruction description has no effect on the order of evaluation.
-
- In general, condition codes are affected on the value of actual stored results, not on *true* results that might be generated internally to greater precision. Thus, for example, two positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the *true* result is clearly positive.
-

Range and Extent Notation

An integer range is specified in English by the word *through*, or in notational form by a double period (...), and is inclusive. For example, the range *0 through 4*, or *0..4*, means the integers 0, 1, 2, 3, and 4.

An extent is given by a pair of numbers separated by a colon and is also inclusive. For example, bits 7:3 specifies an extent of bits including bits 7, 6, 5, 4, and 3.

▪ **MACRO Source Statement Format**

MACRO source statements have four fields—label, operator, operand, and comment fields. The label field defines a location in the program. The operator field specifies the action to be performed. The operator can be a VAX architecture instruction, an assembler directive, or a MACRO call. The operand field contains the instruction operand or operands, the assembler directive argument or arguments, or the MACRO statement or statements. The comment field contains a comment that explains the meaning of the statement. Comments do not affect program execution.

The label and comment fields are optional. The label field must end with a colon (:). The comment field must begin with a semicolon (;). The operand field must conform to the format of the instruction, directive, or MACRO specified in that field. See Figure 5-1 for the MACRO source statement format. The statement format and fields are fully described in the *VAX-11 MACRO Reference Manual*.

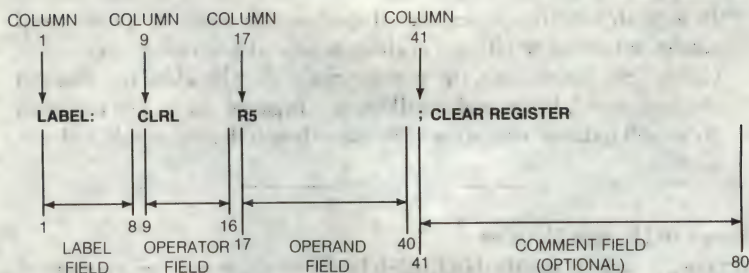


Figure 5-1 ■ MACRO Source Statement Format

Because of printing restrictions, the instruction examples in this book do not conform to the field requirements of the assembler. In practice, the instructions must be formatted as shown in Figure 5-1. A single statement can be continued on several lines by using a hyphen as the last nonblank character before the comment field. When there are no comments, the line can be continued by using a hyphen at the end of the line.

■ Instruction Format

The VAX instruction set has a variable length instruction format whose length depends on the type of instruction. The general instruction format is shown in Figure 5-2. Each instruction consists of an operator followed by up to six operands. The number and type of operands depend on the operator. All operands have the same format; that is, an address mode plus additional information. This additional information contains up to two register designators and addresses, data, or displacements. Operand use is determined implicitly from the opcode and is called the operand type. It includes both the access type and the data type. The example in Figure 5-3 shows several VAX instruction formats.

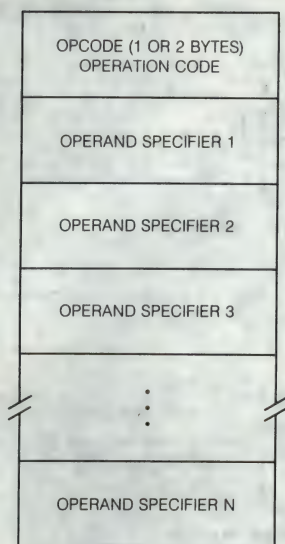


Figure 5-2 ■ General VAX Instruction Format

MOVL 6(R1), R5 ; SIX IS ADDED TO R1. THE RESULT USED AS AN
 ; ADDRESS AND THE CONTENTS OF THAT ADDRESS
 ; IS MOVED TO R5

BYTE

1	MOVL	OPCODE
2	(R1)	} OPERAND SPECIFIER 1
3	6	
4	R5	OPERAND SPECIFIER 2

A. MOVE LONG INSTRUCTION

MOVW#1×3456, -(SP) ; THE NUMBER 3456 IS PUSHED ON THE
 ; STACK

BYTE

1	MOVW	OPCODE
2	(PC) +	OPERAND SPECIFIER 1
3	56	} IMMEDIATE DATA (56 STORED IN BYTE 3) (34 STORED IN BYTE 4)
4	34	
5	-(SP)	OPERAND SPECIFIER 2

B. MOVE WORD INSTRUCTION

ADDL 3 (SP) +, R4, R5 ; NUMBER ON THE STACK IS
 ; ADDED TO THE CONTENTS OF
 ; R4 AND RESULT IS STORED
 ; IN R5

BYTE

1	ADDL 3	OPCODE
2	(SP) +	OPERAND SPECIFIER 1
3	R4	OPERAND SPECIFIER 2
4	R5	OPERAND SPECIFIER 3

C. ADD LONG INSTRUCTION (3 OPERAND)

Figure 5-3 ■ Instruction Formats

Operator Field

Each VAX instruction contains an operating code (opcode) that specifies the operation to perform. An instruction is specified by the byte address of its opcode. The opcode may be one or two bytes long depending on the contents of the byte at address A. Two bytes are used under the condition that the value of the first byte is FD (hexadecimal) through FF (hexadecimal). Figure 5-4 illustrates the opcode formats.

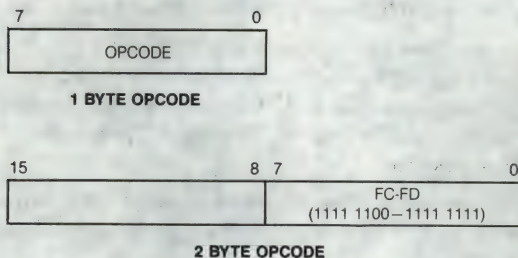


Figure 5-4 • Opcode Format

Operand Field

The operand field contains an operand specifier that gives the information needed to locate the operand. Each general mode addressing description includes the definition of the operand address and the specified operand. For operand specifiers of address access type, the operand address is the actual instruction operand. For other access types, the specified operand is the instruction operand. The branch mode addressing description includes the definition of the branch address.

The operand types specify how the operand is to be used. The opcode information includes the data type of each operand and how the operand is accessed. The data types include byte, word, longword, quadword, octaword, and all the floating types. The following groups of data types are considered equivalent within groups for addressing mode considerations:

-
- Longword and F__ floating
-
- Quadword, D__ floating, and G__ floating
-
- Octaword and H__ floating
-

An operand may be accessed in one of six ways.

1. Read—the specified operand is read-only.
2. Write—the specified operand is write-only.
3. Modify—the specified operand is read, potentially modified, and is written. This is not a memory interlock.

4. Address—the address of the operand in the form of a longword is the actual instruction operand. The operand is not accessed directly although the instruction may subsequently use the address to access that operand.
5. Variable bit field base address—same as address access type except for register mode. In register mode, the field is stored in the register designated by the destination operand or in the destination register concatenated with the next higher addressed register. This access type is a special variant of the address access type.
6. Branch—no operand is accessed. The operand specifier itself is a branch displacement. In this specifier, the data type indicates the size of the branch displacement.

For the address and branch address type that do not directly reference operands, the data type indicates the address and branch. The address indicates the operand size to be used in the address calculation in the autoincrement, autodecrement, and index modes. The branch indicates the branch displacement.

■ Addressing Modes

VAX register addressing can be divided into two broad categories—general mode addressing and branch addressing. The sections that follow describe the various modes under both categories.

Table 5-2 contains a summary of the general register and program counter addressing modes. It shows the mode specifier for each addressing mode in hexadecimal and decimal notation; the assembler notation; the access types that may be used with the various modes; the effect on the program and stack pointer; and which modes may be indexed. For example, in literal mode, only a read access may occur. Any other type of access results in a fault. The program counter and stack pointer are not referenced in this mode and are logically impossible. If indexing is attempted in this mode, a reserved addressing mode fault occurs.

Table 5-2 • Addressing Modes

GENERAL REGISTER ADDRESSING

Hex	Dec	Name	Assembler	r	m	w	a	v	PC	SP	Indexable?
0-3	0-3	literal	S↑#literal	y	f	f	f	f	li	li	f
4	4	indexed	i[Rx]	y	y	y	y	y	f	y	f
5	5	register	Rn	y	y	y	f	y	u	uq	f
6	6	register deferred	(Rn)	y	y	y	y	y	u	y	y
7	7	autodecrement	-(Rn)	y	y	y	y	y	u	y	ux
8	8	autoincrement	(Rn) +	y	y	y	y	y	p	y	ux
9	9	autoincrement deferred	@(Rn) +	y	y	y	y	y	p	y	ux
A	10	byte displacement	B↑D (Rn)	y	y	y	y	y	p	y	y
B	11	byte displacement deferred	@B↑D (Rn)	y	y	y	y	y	p	y	y
C	12	word displacement	W↑D (Rn)	y	y	y	y	y	p	y	y
D	13	word displacement deferred	@W↑D (Rn)	y	y	y	y	y	p	y	y
E	14	longword displacement	L↑D (Rn)	y	y	y	y	y	p	y	y
F	15	longword displacement deferred	@L↑D (Rn)	y	y	y	y	y	p	y	y

PROGRAM COUNTER ADDRESSING

8	8	immediate	I↑# constant	y	u	u	y	y	li	li	y
9	9	absolute	@# address	y	y	y	y	y	li	li	y
A	10	byte relative	B↑address	y	y	y	y	y	li	li	y
B	11	byte relative deferred	@B↑address	y	y	y	y	y	li	li	y
C	12	word relative	W↑address	y	y	y	y	y	li	li	y
D	13	word relative deferred	@W↑address	y	y	y	y	y	li	li	y
E	14	longword relative	L↑address	y	y	y	y	y	li	li	y
F	15	longword relative deferred	@L↑address	y	y	y	y	y	li	li	y

Legend

a	=	address access
D	=	displacement
f	=	reserved addressing mode fault
i	=	any indexable addressing mode
li	=	logically impossible
m	=	modify access
p	=	program counter addressing
r	=	read access
u	=	unpredictable
uo	=	unpredictable for octaword and H_floating format only
uq	=	unpredictable for quadword, octaword, D_floating, G_floating, and H_floating (and field, if position + size is greater than 32)
ux	=	unpredictable for index register same as base register
v	=	field access
w	=	write access
y	=	yes, always valid addressing mode

General Mode Addressing

In general mode addressing, there are two types of addressing—general register addressing and program counter addressing. General register addressing has nine modes while program counter addressing has four.

▪ *General Register Addressing*

The nine modes in which to access general registers are autodecrement, autoincrement, autoincrement deferred, displacement, displacement deferred, index, literal, register, and register deferred. Each is described in the ensuing paragraphs.

Autodecrement Mode. With autodecrement mode, the size of the operand in bytes is subtracted from the content of specified source register. Then the content of the destination register is replaced by the remainder of the subtraction. The remainder is the address of the operand.

To specify the autodecrement mode, the source register operand is enclosed in parentheses and is preceded by a minus (–) sign. See Example 5-1 for the format of this address mode.

Example 5-1 • Autodecrement Mode Instruction

MOVL -(R3), R4

Figure 5-5 shows a *move long* instruction using autodecrement mode. The contents of register R3 are decremented according to the data type specified in the opcode. In this example, the register contents are decremented by 4 because a longword is used. The updated contents of R3 are then used as the address of the operand. The instruction causes the operand to be fetched and loaded into register R4.

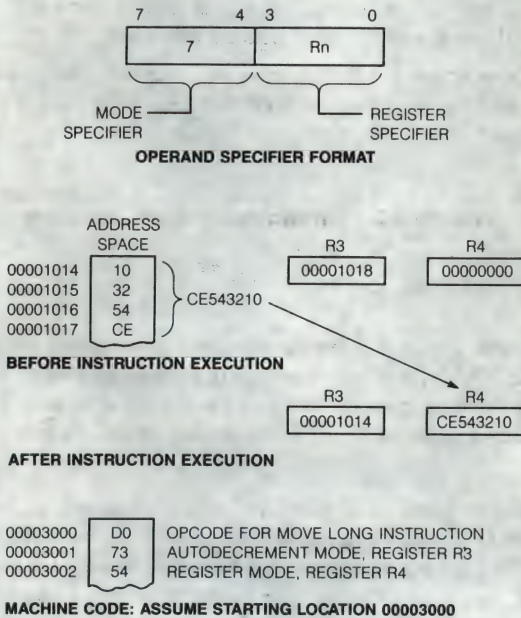


Figure 5-5 • Autodecrement Mode Instruction

The program counter may not be used in autodecrement mode. If it is, the address of the operand is *unpredictable* and the next instruction executed or the next operand specifier is *unpredictable*.

Autoincrement Mode. In autoincrement mode addressing, the register specified in the source register operand contains the address of the operand. After the operand address is determined, the size of the operand is added to the contents of the source register. Then the contents of the destination register are replaced by the sum of the addition. This mode provides for automatic stepping of a pointer through sequential elements of a table of operands. Contents of registers are incremented to address the next sequential location.

The autoincrement mode is especially useful for array processing and stacks. It accesses an element of a table and then steps the pointer to address the next operand in the table. Although most useful for table handling, this mode is general and may be used for variety of purposes.

If the program counter is used as the general register, this addressing mode is considered immediate mode and has special syntax. Immediate mode is described in the section on Program Counter Addressing.

The autoincrement mode is specified by enclosing the register identifier in parentheses followed by a plus (+) sign. See Example 5-2 for the format of the instruction.

Example 5-2 • Autoincrement Mode Instruction

```
MOVL      (R1)+,R2
```

Figure 5-6 shows a *move long* instruction using autoincrement mode. The content of register R1 is the effective address of the source operand. Because the operand is a 32-bit longword, 4 bytes are transferred to register R2. Register R1 is then incremented by 4 because the instruction specifies a longword data type.

Autoincrement Deferred Mode. In autoincrement deferred addressing, the source register contains a longword address that is a pointer to the operand address. After the operand address has been determined, 4 is added to the contents of the source register. The contents of the source register are replaced with the sum of the addition. The quantity 4 is used because there are 4 bytes in an address.

Autoincrement deferred mode is specified by an *at* (@) sign, the source register enclosed in parentheses, followed by a plus (+) sign. Example 5-3 contains a register in the autoincrement deferred mode.

Example 5-3 • Autoincrement Deferred Mode Instruction

MOVW @(R1)+, R2

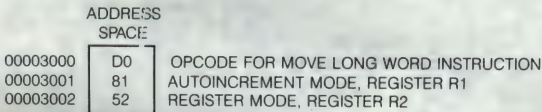
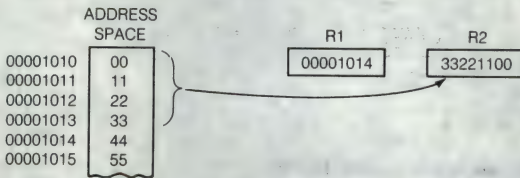
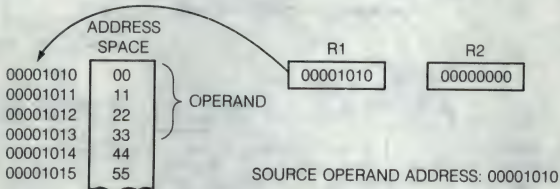
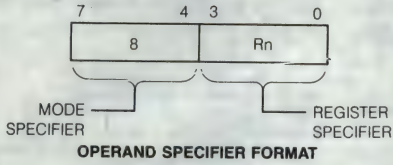


Figure 5-6 • Autoincrement Mode Instruction

Figure 5-7 shows a *move word* instruction using autoincrement deferred mode. Register R1 is a pointer to the operand address. Because a word length instruction is specified, the byte at the effective address and the byte at the effective address plus 1 are loaded into the low-order half of register R2. The upper half of register R2 is unaltered. Register R1 is then incremented by 4 since it points to a 32-bit address.

If the program counter is used as the general register, this addressing mode is considered absolute mode. Absolute mode is described in the section on Program Counter Addressing.

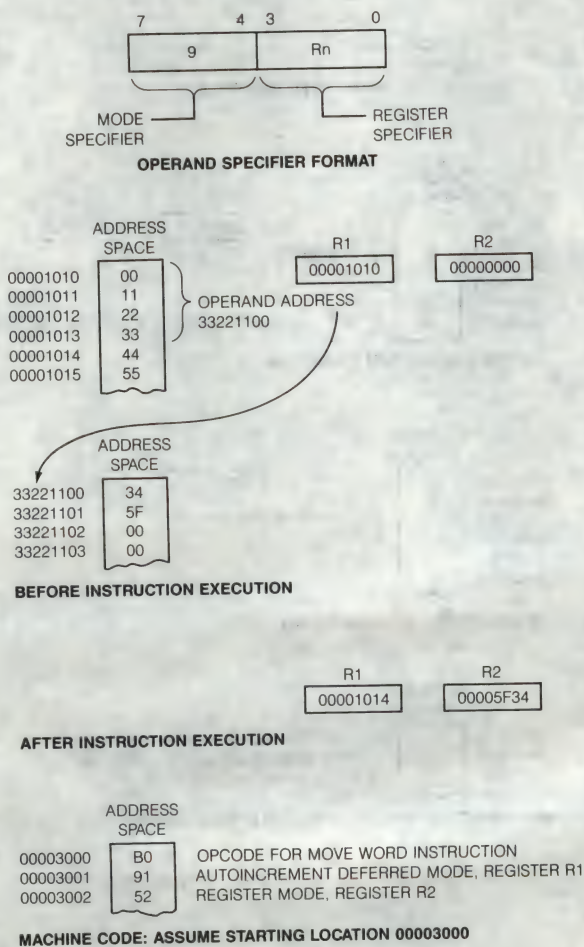


Figure 5-7 • Autoincrement Deferred Mode Instruction

Displacement Mode. The VAX architecture provides for an 8-bit, 16-bit, or 32-bit offset. Because most program references occur within small discrete portions of the address space, a 32-bit offset is not always necessary. The 8- and 16-bit offsets use fewer bits. If the displacement is a byte or a word, it is sign-extended to 32 bits. Then the displacement is added to the content of the specified register. The result is the operand address. See Example 5-4 for the syntax of the displacement mode.

Example 5-4 • Byte Displacement Mode Instruction

MOVB B↑5(R4), B↑3(R3)

Figure 5-8 shows a move byte instruction using displacement mode. A displacement of 5 is added to the content of R4 to form the address of the byte operand. The operand is moved to the address formed by adding the displacement of 3 to the contents of R3.

Three data types can be specified. For example,

-
- B↑*d*(R*n*) forces byte displacement.

 - W↑*d*(R*n*) forces word displacement.

 - L↑*d*(R*n*) forces longword displacement.

If the program counter is used as the general register, this mode is called relative mode. The Program Counter Register Addressing section describes the relative mode.

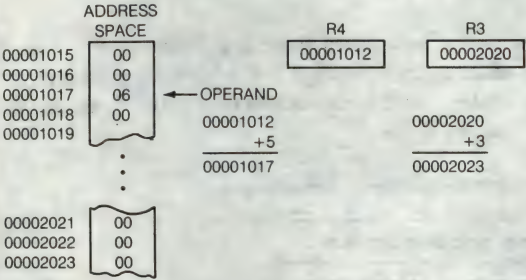
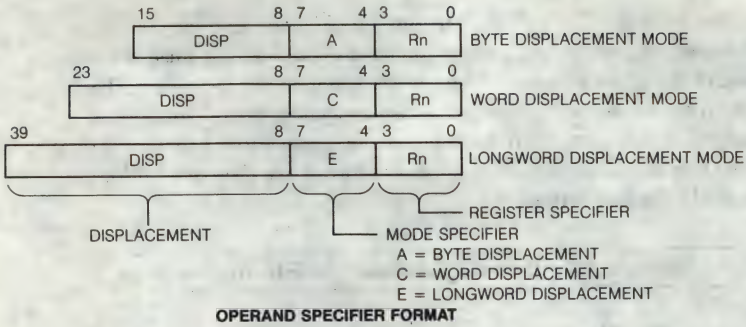
Displacement Deferred Mode. If the displacement is a byte or word, it is sign-extended to 32 bits. Then the displacement is added to the contents of the selected general register. The result is a longword address of the operand address. See Example 5-5 for an example of an instruction in displacement deferred mode.

Three data types can be specified. For example,

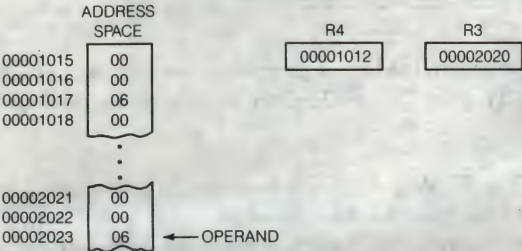
-
- @B↑*d*(R*n*) forces byte displacement deferred mode.

 - @W↑*d*(R*n*) forces word displacement deferred mode.

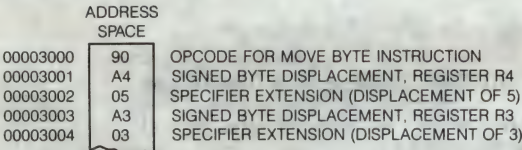
 - @L↑*d*(R*n*) forces longword displacement deferred mode.



BEFORE INSTRUCTION EXECUTION



AFTER INSTRUCTION EXECUTION



MACHINE CODE: ASSUME STARTING LOCATION 00003000

Figure 5-8 • Displacement Mode Instruction

Example 5-5 • Byte Displacement Deferred Mode Instruction

INCW @B15(R4)

Figure 5-9 shows an increment word instruction using displacement deferred mode. The quantity 5 is added to the contents of register R4 to produce the longword address of the address of the operand. The operand of 5713 is incremented to 5714.

If the program counter is used as the general register, this is considered *relative deferred mode*.

Index Mode. Index mode addressing provides very general and efficient accessing of arrays. The VAX architecture provides for context indexing where the number in the index register is shifted left by the context of the data type specified. It is not shifted for byte data, shifted once for word data, twice for longword data, three times for quadword data, and four times for octaword data. This allows loop control variables to be used in the address calculation without first shifting them the appropriate number of times. This minimizes the number of instructions required. This feature is used to advantage in the FORTRAN VAX compiler.

The operand specifier consists of at least two bytes—a primary operand specifier and a base operand specifier. The primary operand specifier contained in bits 0 through 7 includes the index register (Rx) and a mode specifier of 4. The address of the primary operand is determined by first multiplying the contents of index register Rx by the size of the primary operand in bytes. This value is then added to the address specified by the base operand specifier (bits 15:8), and the result is taken as the operand address.

Specifying register, literal, or index mode for the base operand specifier results in an illegal addressing mode fault. If the use of some particular specifier is illegal (that is, causes a fault or unpredictable behavior), then that specifier is also illegal as a base operand specifier in index mode under the same conditions.

The following restrictions are placed on index register Rx:

1. The program counter cannot be used as an index register. If it is, a reserved addressing mode fault occurs.
2. If the base operand specifier is for an addressing mode that modifies a register, that register cannot be the index register. If it is, the primary operand address is *unpredictable*. Addressing modes that modify a register are the autoincrement, autoincrement deferred, and autodecrement modes.

Table 5-3 lists the various forms of index mode addressing available. The names of the addressing modes resulting from index mode addressing are formed by adding *index* to the addressing mode of the base operand specifier. The general register is designated *Rn* and the indexed register is *Rx*.

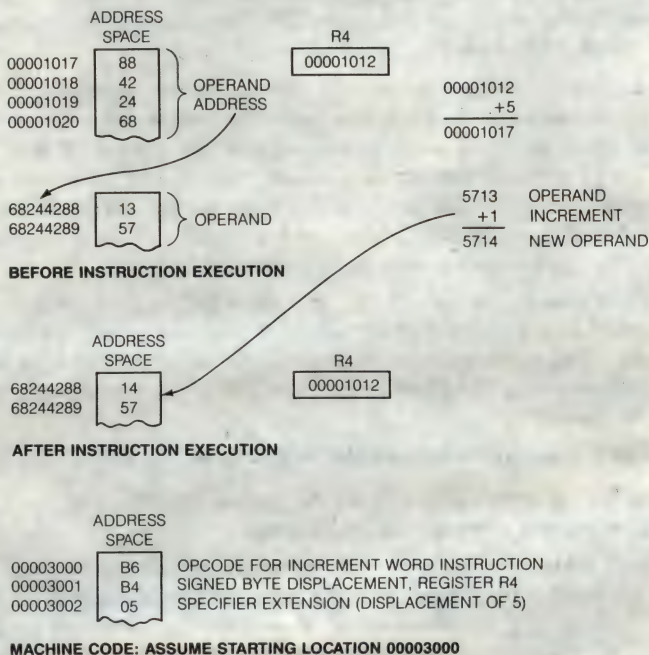
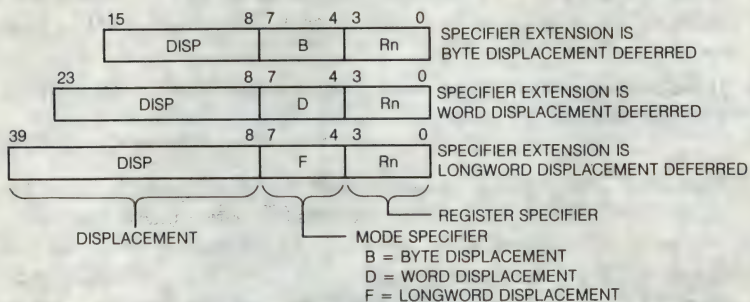


Figure 5-9 • Displacement Deferred Mode Instruction

Table 5-3 ■ Index Mode Addressing

Index Mode	Assembler Notation
Absolute	@#address [Rx]
Autodecrement	-(Rn) [Rx]
Autoincrement	(Rn) + [Rx]
Autoincrement Deferred	@(Rn) + [Rx]
Deferred Displacement:	
Byte	@B↑D(Rn) [Rx]
Word	@W↑D(Rn) [Rx]
Longword	@L↑D(Rn) [Rx]
Immediate ¹	I↑# constant [Rx]
Immediate Displacement:	
Byte	B↑D(Rn) [Rx]
Word	W↑D(Rn) [Rx]
Longword	L↑D(Rn) [Rx]
Register Deferred	(Rn) [Rx]
Relative Indexed	address [Rx]

¹ The instruction is recognized by assembler but is not generally useful. The operand address is independent of the value of the constant.

It is important to note that the operand address (the address containing the operand) is first evaluated. Then the index specified by the index register is added to the operand address to find the indexed address. To illustrate this, an example of each type of indexed addressing is shown in Examples 5-6 through 5-12.

Register Deferred Index Mode. See Example 5-6.

Example 5-6 ■ Register Deferred Index Mode Instruction

INCW (R2) [R5]

Figure 5-10 shows an increment word instruction using register deferred index addressing. The base operand address is evaluated. This location is indexed by 6 because the value (3) in the index register is multiplied by the word data size of 2.

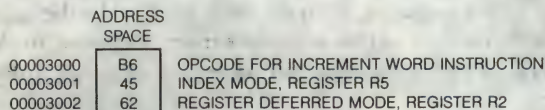
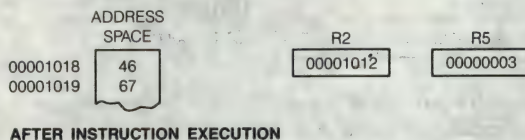
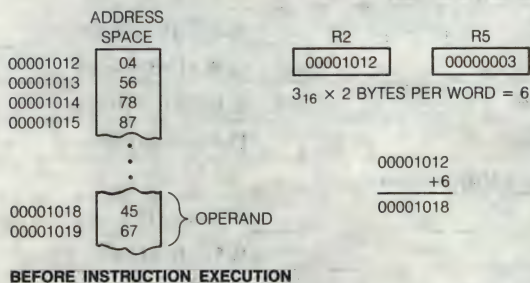
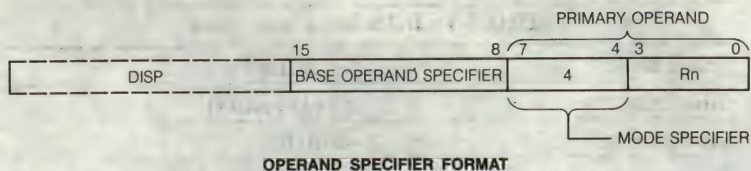


Figure 5-10 • Register Deferred Index Mode Instruction

Autoincrement Index Mode. See Example 5-7.

Example 5-7 • Autoincrement Index Mode Instruction

CLRL (R4)+[R5]

Figure 5-11 shows a *clear longword* instruction using the autoincrement indexed addressing mode. The base operand address is in register R4. This value is indexed by the quantity in R5 multiplied by the data size. This location, plus the next three, are cleared because a *clear longword* instruction is specified.

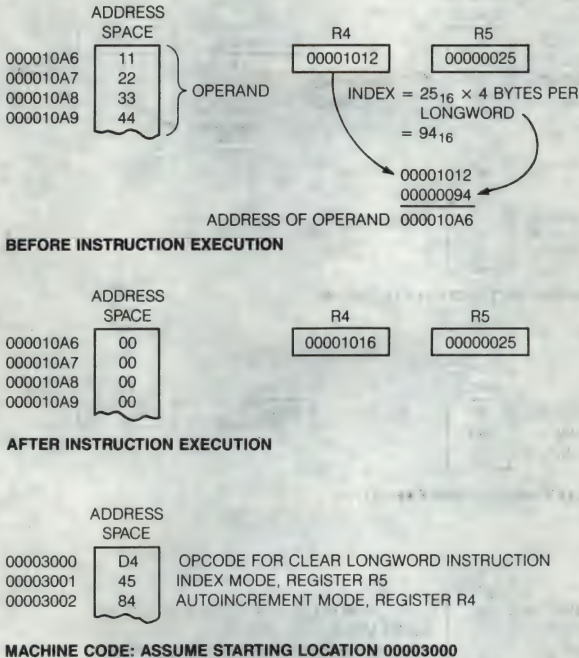


Figure 5-11 • Autoincrement Index Mode Instruction

Autoincrement Deferred Index Mode. See Example 5-8.

Example 5-8 • Autoincrement Deferred Index Mode Instruction

CLRW @(R4)+[R5]

Figure 5-12 shows a *clear word* instruction using the autoincrement deferred indexing mode. Register R4 contains the address of the operand address. The index value A is obtained by multiplying the contents (5) of the index register by the context of the data type, which is 2. The calculated word address is cleared.

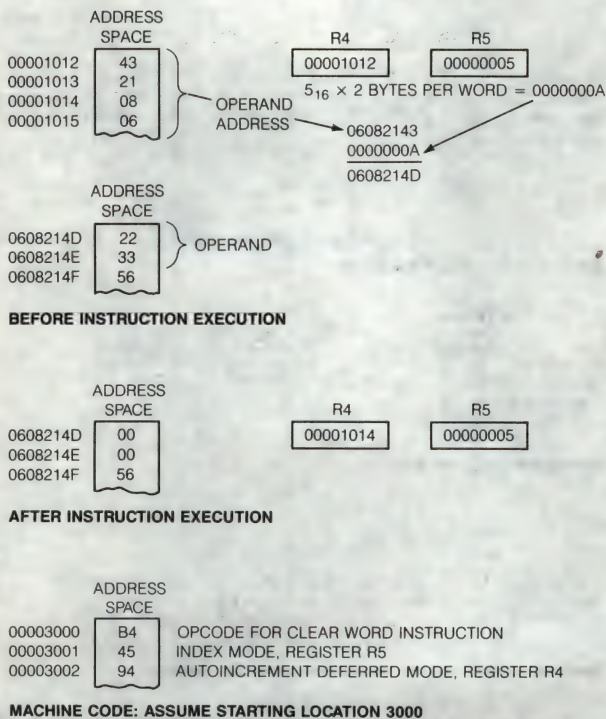


Figure 5-12 • Autoincrement Deferred Index Mode Instruction

Autodecrement Index Mode. See Example 5-9.

Example 5-9 • Autodecrement Index Mode Instruction

CLRW -(R2) [R4]

Figure 5-13 shows a *clear word* instruction using autodecrement indexed mode. Register R2 is predecremented and the indexed value is calculated as 6. Because a *clear word* instruction is specified, two bytes are cleared.

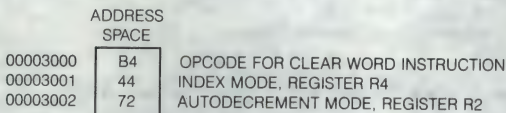
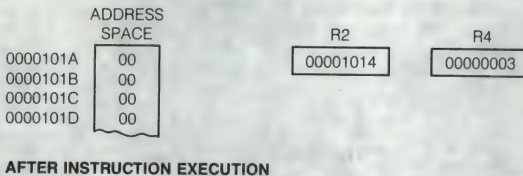
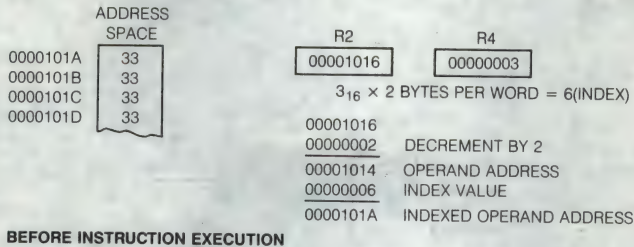


Figure 5-13 ■ Autodecrement Index Mode Instruction

Absolute Index Mode. See Example 5-10.

Example 5-10 ■ Absolute Index Mode Instruction

CLRL @#↑X1012 [R2]

Figure 5-14 shows a clear longword instruction using absolute indexed mode. The base of 00001012 (hexadecimal) is indexed by R2 that contains 5. Because a longword data type is specified, $5 \times 4 = 14$ (hexadecimal), which becomes the index value. This value is added to 00001012 (hexadecimal) yielding 0001026 (hexadecimal). This is the operand address, and four bytes are cleared because a longword data type has been specified.

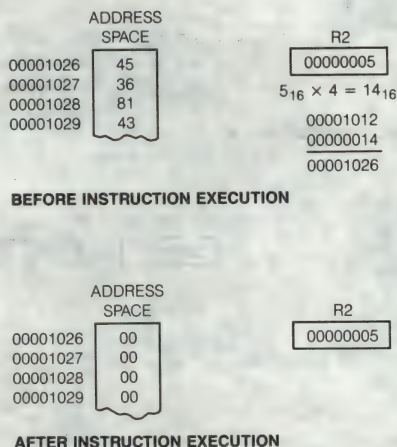


Figure 5-14 • Absolute Index Mode Instruction

Displacement Index Mode. See Example 5-11.

Example 5-11 • Displacement Index Mode Instruction

CLRQ 2(R1)[R3]

Figure 5-15 shows a *clear quadword* instruction using displacement index mode. The byte displacement of 2 is added to the contents of register R1. The index, calculated as 28, is added to this address. Because a quadword was specified, this location and the next seven locations are cleared.

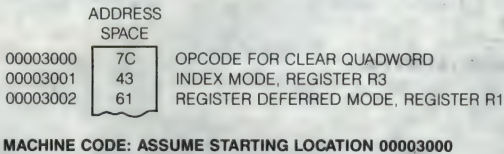
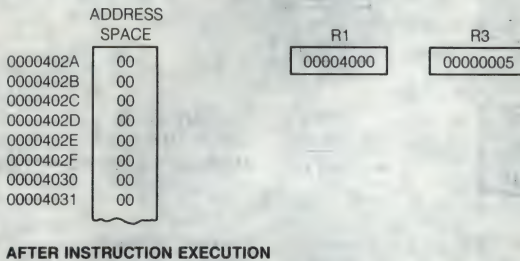
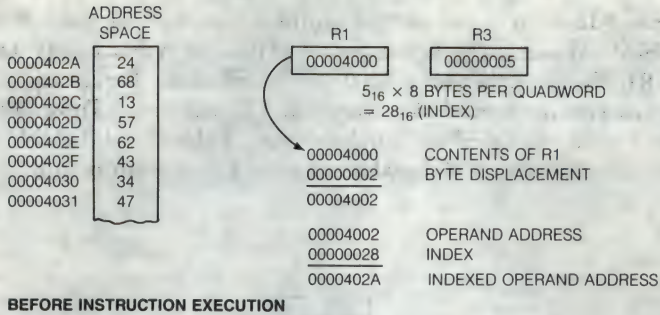


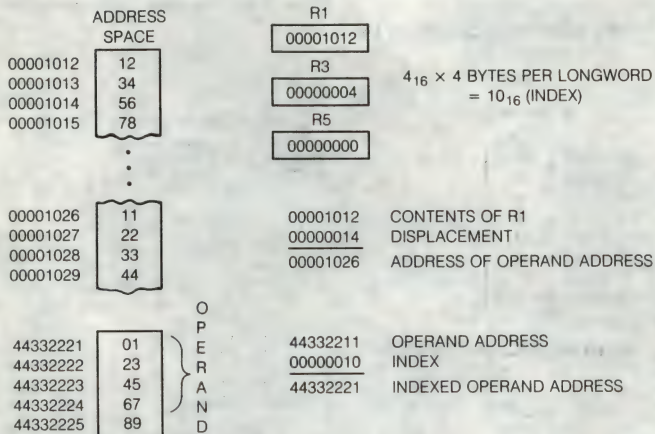
Figure 5-15 ■ Displacement Index Mode Instruction

Displacement Deferred Index Mode. See Example 5-12.

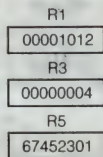
Example 5-12 ■ Displacement Deferred Index Mode Instruction

MOVL @1X14(R1)[R3],R5

Figure 5-16 shows a *move longword* instruction using displacement deferred indexed addressing. The displacement of 14 is added to the contents of register R1. The sum is the address 00001026 (hexadecimal). The contents of this location yield the operand address (44332211 (hexadecimal)). This quantity is added to the index yielding the indexed operand address of 44332221 (hexadecimal). The contents of this address are then moved into register R5.



BEFORE INSTRUCTION EXECUTION



AFTER INSTRUCTION EXECUTION

Figure 5-16 ■ Displacement Deferred Index Mode Instruction

Literal Mode. Literal mode addressing provides an efficient means of specifying integer constants in the range from 0 to 63. This is called *short literal*. Literal values greater than 63 are obtained by using the program counter in autoincrement mode (immediate mode). For predefined values, the assembler chooses between short literal and immediate modes. The format for short literal operands is shown in Figure 5-17. Bits 7 and 6 are always set to zero. Figure 5-18 shows some short literals (14, 30, 46, and 62). To specify literal mode, prefix the literal with $S\uparrow\#$.

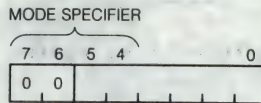


Figure 5-17 ■ Short Literal Operand

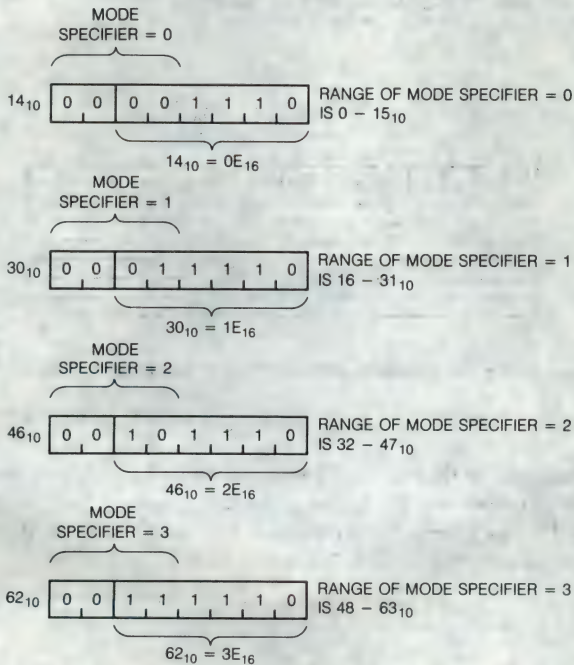


Figure 5-18 ■ Typical Short Literal Operands

Floating-point literals as well as short literals can be expressed. The floating-point literals are listed in Table 5-4. For operands of the short floating type, the 6-bit literal field in the operand specifier is composed of two 3-bit fields. The field marked EXP designates the exponent column and FRAC designates the fraction columns. See Figure 5-19.

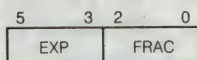


Figure 5-19 • Literal Field

The 3-bit EXP field and 3-bit FRAC field are used to form an F__ floating or D__ floating operand as shown in Figure 5-20. Bits 63:32 are not present in an F__ floating operand. G__ floating and H__ floating operands can be formed in analogous ways using the EXP and FRAC fields.

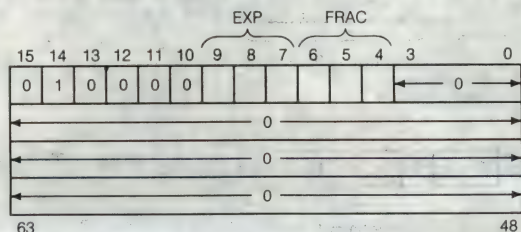


Figure 5-20 • D__ floating and F__ floating Operands in Literal Mode

Bits 3 through 5 of the EXP field are stored in bits 7 through 9, respectively, of the floating operand. (See Figure 5-21.) Bits 0 through 2 of the FRAC field are stored in bits 4 through 6 in the floating operand. The decimal values that can be stored are given in Table 5-4.

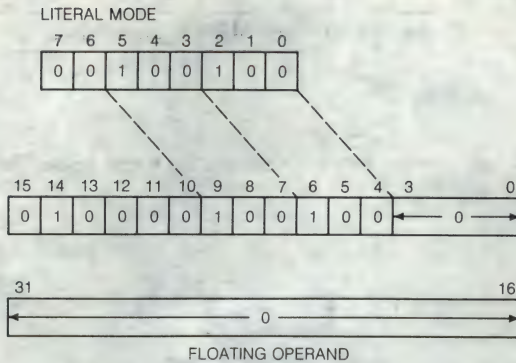


Figure 5-21 ■ Floating Operand Bit Storage

The EXP field is expressed in *excess 128* notation. In this notation, an offset of 128 is added to the exponent. For example, an exponent of 0 is represented as 128 or 10000000 (binary), while an exponent of 3 is represented as 131 or 10000011 (binary).

Assume you want to express the floating-point literal of 64. Find the integer 64 in the table. It is in the 7 row of EXP and the 0 column of the fraction columns. Therefore, 7 is the value of the exponent field and 0 is the value of the fraction field.

Table 5-4 ■ Floating Literals

Exponent	Fraction							
	0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1-1/8	1-1/4	1-3/8	1-1/2	1-5/8	1-3/4	1-7/8
2	2	2-1/4	2-1/2	2-3/4	3	3-1/4	3-1/2	3-3/4
3	4	4-1/2	5	5-1/2	6	6-1/2	7	7-1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

Example 5-13 • Literal Mode Instruction

MOVL S†#9,R4

Figure 5-22 shows a *move long* instruction using literal mode. The literal 9 is transferred to register R4.

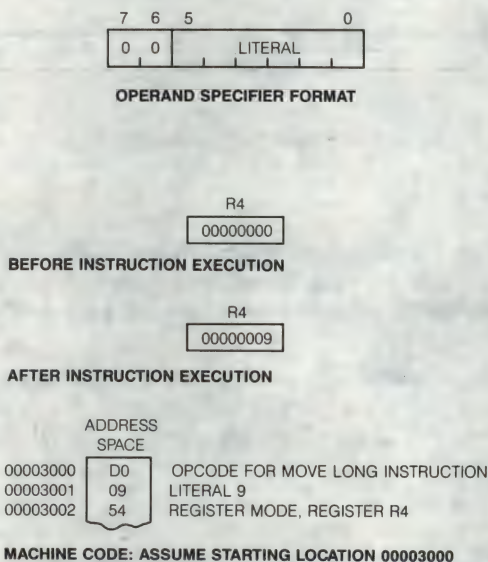


Figure 5-22 • Literal Mode Instruction

Register Mode. With register mode, any of the general registers may be used as simple accumulators and the operand is contained in the selected register. Because they are hardware registers within the processor, they provide speed advantages when used for operating on frequently accessed variables.

This mode can be used with operand specifiers using read, write, or modify access but cannot be used with the address access type. Otherwise, an illegal addressing mode fault occurs. The program counter cannot be used in this mode. If the program counter is read, the value is *unpredictable*. If the program counter is written, the next instruction executed or the next operand specified is *unpredictable*. Similarly, if the program counter is used in register mode for a write-access operand that takes two adjacent registers, the contents of register 0 are *unpredictable*.

If register 12, 13, the stack pointer, or program counter is used in register mode addressing for an operand that takes four adjacent registers, the results are *unpredictable*. If the program counter is used in register mode for a write access that requires four adjacent registers, the contents of registers 0, 1, and 2 are *unpredictable*. Likewise, if register 13 is used in register mode for a write access that takes four adjacent registers, the contents of register 0 are *unpredictable*. If the stack pointer is used in register mode for a write access that takes four adjacent registers, the contents of registers 0 and 1 are *unpredictable*.

The stack pointer cannot be used in this mode for an operand that takes two adjacent registers because that implies a direct reference to the program counter and the results are *unpredictable*.

The operand is the content of register n , or $R[n + 1]$ concatenated with Rn for quadword, D__ floating, and certain field operations. The following list identifies the single- and multiple-register operand format.

One register operand = Rn

Two register operand = $R[n + 1]'R[n]$

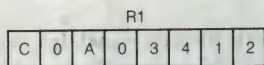
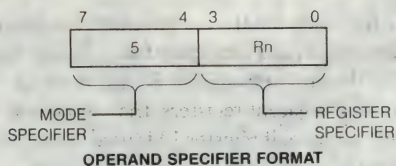
Four register operand = $R[n + 3]'R[n + 2]'R[n + 1]'R[n]$

Example 5-14 • Register Mode Instruction

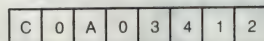
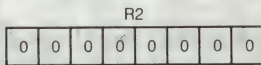
MOVW R1, R2

Figure 5-23 shows a *move word* instruction using register mode. The content of register 1 is the operand. The *move word* instruction transfers the least significant half of register 1 to the least significant half of register 2. The upper half of register 2 is unaffected.

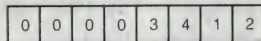
Register Deferred Mode. The register deferred mode provides one level of indirect addressing over register mode. That is, the general register contains the address of the operand rather than the operand itself. The deferred modes are useful when dealing with an operand whose address is calculated. The program counter cannot be used in register deferred mode addressing as the results are *unpredictable*. To indicate the register deferred mode, enclose the register operand in parentheses. See Example 5-15 for the format of the instruction.



BEFORE INSTRUCTION EXECUTION



AFTER INSTRUCTION EXECUTION



00003000	80	OPCODE FOR MOVE WORD INSTRUCTION
00003001	51	OPERAND SPECIFIER, SOURCE; REGISTER MODE 1
00003002	52	OPERAND SPECIFIER, DESTINATION; REGISTER MODE 2

MACHINE CODE: ASSUME STARTING LOCATION 00003000

Figure 5-23 • Register Mode Instruction

Example 5-15 • Register Deferred Mode Instruction

CLRQ (R4)

Figure 5-24 shows a *clear quadword* instruction using register deferred mode. Register 4 contains the address of the operand. The instruction specifies that the byte at this address plus the following seven bytes are to be cleared.

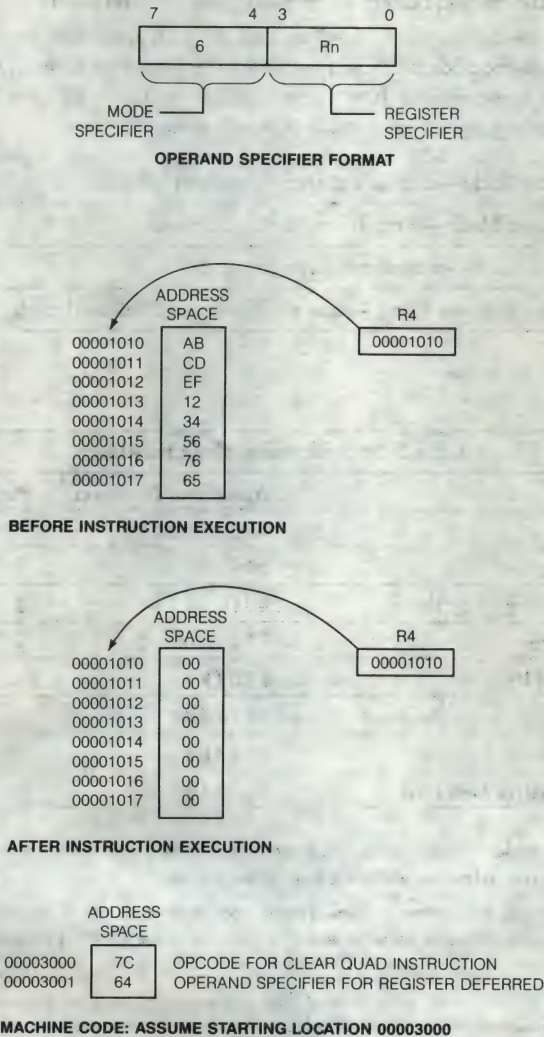


Figure 5-24 ■ Register Deferred Mode Instruction

▪ Program Counter Register Addressing

Register 15 is used as the program counter. It can also be used as a register in addressing modes. The processor increments the program counter as the opcode, operand specifier, and immediate data or address of the instruction are evaluated. The amount that the program counter is incremented is determined by the opcode, number of operand specifiers, and so on.

The program counter can be used with all the VAX addressing modes except register or index mode. In those two modes, the results are *unpredictable*. The addressing mode register functions are shown in Table 5-5. The following modes use the program counter as a general register.

- Absolute Mode—same as autoincrement deferred mode
- Immediate Mode—same as autoincrement mode
- Relative Mode—same as displacement mode
- Relative Deferred Mode—same as displacement deferred mode.

Table 5-5 • Addressing Mode Functions

Mode	Assembler Notation	Note
Absolute	@#Location	*
Byte Relative	B↑G (R)	†
Byte Relative Deferred	@B↑G (R)	‡
Immediate	I↑#Operand	§
Longword Relative	L↑G (R)	
Longword Relative Deferred	@L↑G (R)	
Word Relative	W↑G (R)	
Word Relative Deferred	@W↑G (R)	

* Absolute mode is the same as autoincrement mode with the program counter used as a general register. Absolute address follows address mode.

† Relative mode is the same as displacement mode with the program counter used as a general register. Displacement is added to current value of PC to obtain operand address.

‡ Relative deferred mode is the same as displacement deferred mode with the program counter used as a general register. Displacement is added to current value of PC to yield address of operand address.

§ Immediate mode is the same as autoincrement mode with the program counter used as a general register. The constant operand follows the address mode.

When a standard program is available for different users, it is often helpful to be able to run it at different areas of virtual memory. VAX computers can accomplish the relocation of a program very efficiently through the use of position-independent code. If an instruction and its objects are moved in such a way that the relative distance between them is not altered, the same offset relative to the program counter can be used in all positions in memory.

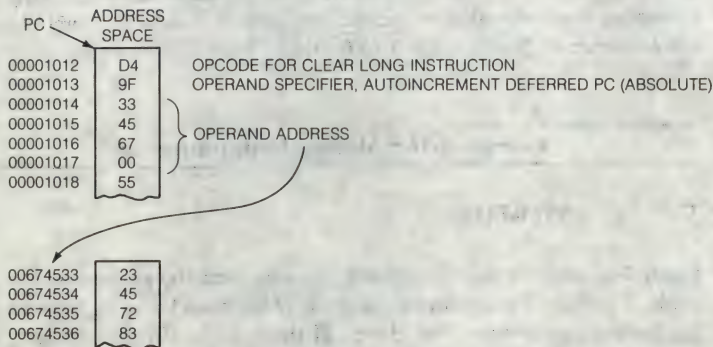
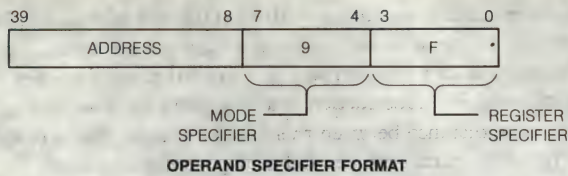
Absolute Mode. This mode is autoincrement deferred when using the program counter as a general register. The contents of the location following the addressing mode are taken as the operand address. This is interpreted as an absolute address. See Example 5-16 for the format of the operand.

Example 5-16 • Absolute Mode Instruction

CLRL @#↑X674533

Figure 5-25 shows a *clear longword* instruction using the absolute addressing mode. This instruction causes the location or locations following the addressing mode to be taken as the address of the operand. In this example, the address is 00674533 (hexadecimal). The longword operand for this address is cleared.

Immediate Mode. The immediate addressing mode is autoincrement mode when the program counter is used as a general register. The contents of the location following the addressing mode are immediate data. Immediate mode may not be used for operands of the modify or write access types. If immediate mode is used for one of those operands, the value of the data read is *unpredictable*. So is the address at which the operand is written. See Example 5-17 for the format of the operand.



BEFORE INSTRUCTION EXECUTION

00674533	00
00674534	00
00674535	00
00674536	00

AFTER INSTRUCTION EXECUTION

Figure 5-25 • Absolute Mode Instruction

Example 5-17 • Immediate Mode Instruction

MOVL #6,R4

Figure 5-26 shows a *move longword* instruction using immediate mode. The immediate data (00000006(hexadecimal)) following the mnemonic and operand specifier are moved to register R4.

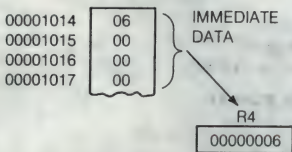
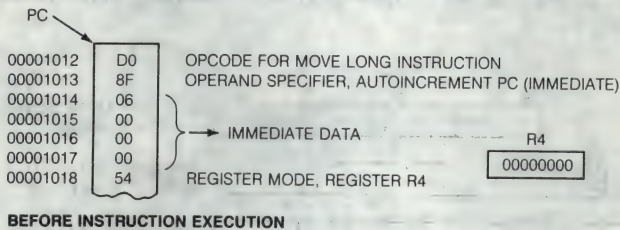
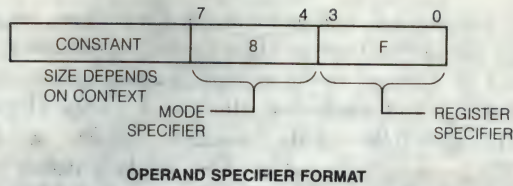


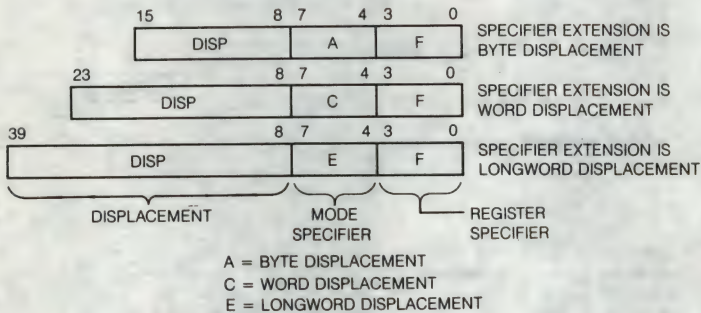
Figure 5-26 • Immediate Mode Instruction

Relative Mode. This mode is the displacement mode with the program counter used as a general register. The displacement follows the operand specifier and is added to the program counter. The sum of which becomes the address of the operand. This mode is useful for writing position-independent code because the location referenced is always fixed. See Example 5-18 for the format of the operand.

Example 5-18 • Relative Mode Instruction

MOVL ↑X2016, R4

Figure 5-27 shows a *move longword* instruction using relative mode. The word following the address mode is added to the program counter to obtain the address of the operand. In this example, the program counter is pointing to location 00001016 (hexadecimal) after the first operand specifier is evaluated. The word following the mnemonic and first operand specifier is 00001000 (hexadecimal), and is added to the program counter yielding 00002016 (hexadecimal). This value represents the address of the longword operand (00860077 (hexadecimal)). Then this operand is moved to register R4. The program counter contains 00001017 (hexadecimal) after instruction execution.



OPERAND SPECIFIER FORMAT

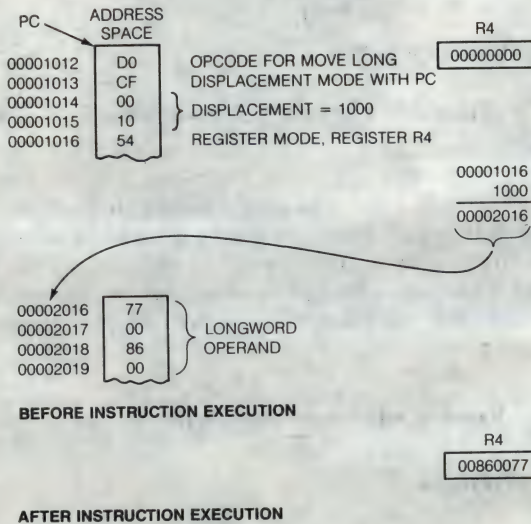


Figure 5-27 • Relative Mode Instruction

Relative Deferred Mode. This mode is similar to relative mode except that the displacement following the addressing mode is added to the program counter. The updated contents of the program counter are the address of the first byte beyond the specifier extension. This addressing mode is useful when processing tables of addresses. See Example 5-19 for the format of the operand.

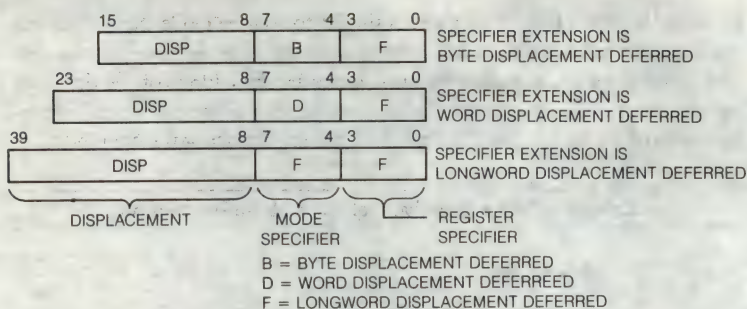
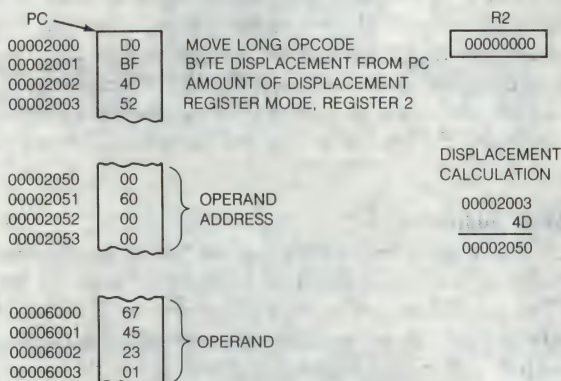
Example 5-19 • Relative Deferred Mode Instruction

MOVL @1X2050,R2

Figure 5-28 shows a move long instruction where 00002050 (hexadecimal) represents the address of the operand. A byte displacement is selected by the assembler because the displacement is within 128 addressable bytes. When the displacement is evaluated, the program counter is pointing to 00002003 (hexadecimal). The displacement of 4D is added to the current value of the program counter yielding the address of 00002050 (hexadecimal). Then the contents of this address are used as the effective operand address (00006000 (hexadecimal)), and the operand of 1234567 (hexadecimal) is moved to register R2.

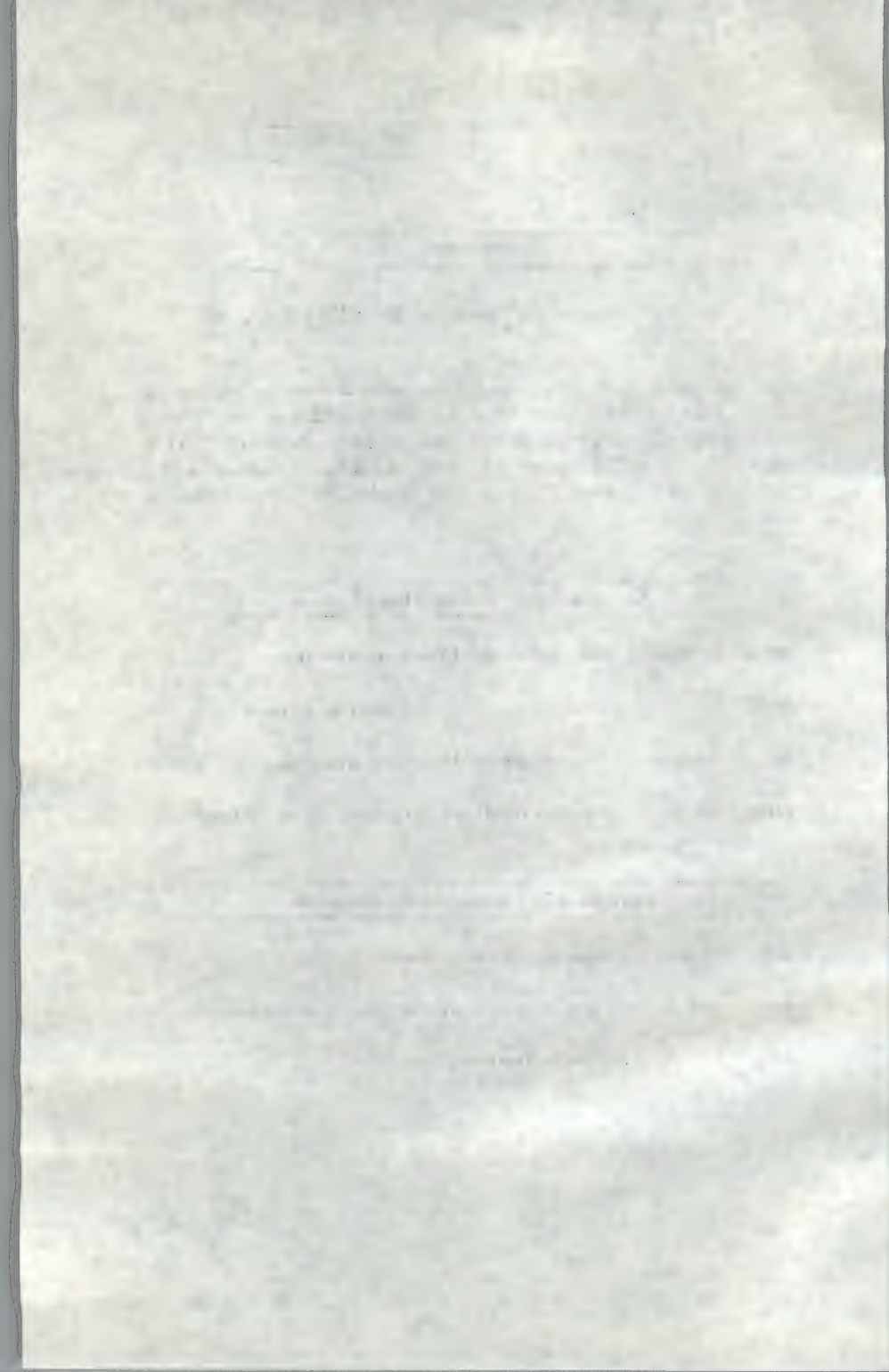
Branch Mode Addressing

In branch mode displacement addressing, the byte or word displacement is sign-extended to 32 bits and added to the updated content of the program counter. The updated content of the program counter is the address of the first byte beyond the operand specifier. The assembler notation for byte and word branch displacement addressing is A where A is the branch address. Note that the branch address and not the displacement is used. See Figure 5-29 for the branch mode instruction operand specifier format.

**OPERAND SPECIFIER FORMAT****BEFORE INSTRUCTION EXECUTION****AFTER INSTRUCTION EXECUTION**

R2
01234567

Figure 5-28 ■ Relative Deferred Mode Instruction



Chapter 6 ■ Functions of the Instruction Set

A major goal of the VAX architecture is to provide an instruction set that is symmetrical with respect to data types. For example, there are separate *add* instructions for seven integer and floating-point data types. Each is available in both two-operand and three-operand format. Other symmetrical operations include data movement, data conversion, data testing, and computation. Thus both assembly language programmers and compilers can choose the best instruction to use independent of the data type.

Instruction mnemonics are formed by combining an base operation abbreviation with a data-type suffix. Conversion instructions are formed by adding suffixes for both the source and destination data types. For example, the basic *convert* instruction is *CVT*. To convert G__ floating to F__ floating, one must affix a *G* for the source and an *H* for the destination data type. This forms the mnemonic *CVTGH*.

Computation instructions have an additional suffix to indicate the choice between two- and three-operand instructions. For example, the *multiply word* instruction uses the mnemonic *MULW*. A two-operand instruction uses *MULW2* and a three-operand instruction uses *MULW3*.

Special instruction mnemonics have been chosen for similarity. For example, a *move word* instruction has the mnemonic *MOVW*, while a *move F__ floating* instruction has the mnemonic *MOVE*. Some instructions span several categories. For example, the *compare* instruction is found in character string, decimal string, floating point, integer, and variable length bit field instructions. Chapter 9 contains detailed descriptions of each instruction. This chapter describes the general functioning of the types of instructions.

Instructions are described in this chapter according to categories. They are Address, Arithmetic, Character String, Control, Cyclic Redundancy Check, Decimal String, Edit, Floating Point, Index, Integer, Logic, Multiple Register, Privileged, Procedure Call, Processor Status Longword, Queue, and Variable Length Bit Field.

■ Address Instructions

Address instructions are used to manipulate addresses. There are two basic address instructions: *move address* (*MOVA*) and *push address* (*PUSHA*). The *move address* instruction replaces one address with another. *Push address* instructions write an address onto a stack.

There are suffixes for each type of data. The suffix B is used to specify byte data, D for D__ floating data, F for F__ floating data, G for G__ floating data, L for longword data, O for octawords, Q for quadwords, and W for words. In order to move an address in F__ floating data, the mnemonic MOVA would have an F affixed to it forming the instruction MOVAF. Other mnemonics are similarly constructed.

■ Arithmetic Instructions

Arithmetic instructions are *add*, *subtract*, *multiply*, and *divide*. The instructions are available in both two- and three-operand forms for each applicable data type. As input, the three-operand form takes the values of the first two operands, performs the operation, and stores the result in the third operand.

■ Character String Instructions

The character string instructions are

-
- *Compare character* (CMPC).

 - *Locate character* (LOCC).

 - *Match character* (MATCHC).

 - *Move character* (MOVC).

 - *Move translated characters* (MOVTC).

 - *Move translated until character* (MOVTUC).

 - *Scan characters* (SCANC).

 - *Skip characters* (SKPC).

 - *Span characters* (SPANC).

A character string is specified by two operands—an unsigned word operand giving the length of the character string in bytes and the address of the lowest addressed byte of the character string. This is specified by a byte operand of address access type.

Each of the character string instructions uses general registers to store a control block that maintains updated addresses and state information during the execution of the instruction. At completion, these registers are available to software to use as string specification operands for a subsequent instruction. During the execution of the instructions, *pending interrupt* conditions are tested. If any are found, the control block is updated, the *first part done* bit of the processor status longword is set, and the instruction is interrupted. After the interruption, the instruction resumes transparently. The format of the control block is shown in Figure 6-1.

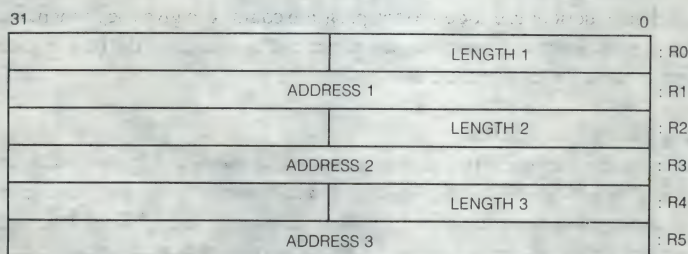


Figure 6-1 ■ Control Block Format

The fields LENGTH 1, LENGTH 2, and LENGTH 3 (if required) contain the number of bytes remaining to be processed in the first, second, and third string operands, respectively. The fields ADDRESS 1, ADDRESS 2, and ADDRESS 3 (if required) contain the address of the next byte to be processed in the first, second, and third string operands, respectively.

■ Control Instructions

Control instructions include case, loop, subroutine, and transfer instructions. In most situations, execution speed is improved if the target of a control instruction is on an aligned longword boundary. But this is not a requirement.

Case Instructions

Dispatching to a routine based on the value of a variable occurs frequently enough that some high-level languages include special constructs to handle it; for example, the computed GOTO in FORTRAN and the *case* statement in PASCAL. Because of this, the VAX instruction set includes a *case* instruction so that such control structures can be represented efficiently. Not only does *case* handle the transfer of control but it also handles the initialization and bounds checking for the *index* variable.

The objective of the *case* instruction is to transfer control to one of several locations based on the value of the *integer selector* operand. The base operand specifies the lower bound for selector. Following the *case* instruction is a table of word displacements for the branch locations. Just as the displacements in branch instructions are added to the program counter to give the branch destination, these word displacements are added to the address of the first displacement to form the case branch destinations.

Loop Control Instructions

There are three loop control instructions—*add compare and branch* (ACB), *add one and branch* (AOB), and *subtract one and branch* (SOB). The instructions efficiently implement the general FOR or DO loops in high-level languages. Specified operands are manipulated and if certain results are obtained, the program counter is loaded with the result of the manipulation.

The *add compare and branch* instruction can accommodate seven types of data—byte, word, longword, D__ floating, F__ floating, G__ floating, and H__ floating. The *add one and branch* instruction adds a one to the specified index operand. The sum of the operation replaces the operand. The *subtract one and branch* instruction removes a one from the specified index operand. The remainder of the operation replaces that operand.

Subroutine Call Instructions

Two special types of *branch* and *jump* instruction are provided for calling subroutines—*branch to subroutine* and *jump to subroutine*. Both instructions save the contents of the program counter on the stack before loading the counter with the new address. With *branch to subroutine* instructions, you can supply either a byte or word displacement.

This shortcut to subroutine calling is complemented by the *return from subroutine* instruction. The instruction removes the first longword of the stack and loads it into the program counter. Because the *branch to subroutine* instruction is either two or three bytes long and the *return from subroutine* instruction is one byte long, extremely efficient programs can be written using subroutines.

The *breakpoint fault* instruction is used in conjunction with the *trace* bit to implement debugging facilities.

Transfer Instructions

The two basic types of control transfer instructions are *branch* and *jump* instructions. Both *branch* and *jump* load new addresses in the program counter. With *branch* instructions, you supply a displacement (offset) that is added to the program counter to obtain the new address. With *jump* instructions, you supply the address you want loaded, using one of the normal addressing modes.

Because most transfers are to locations relatively close to the current instructions, and because *branch* instructions take less space than *jump* instructions, the processor offers a variety of *branch* instructions to choose from. There are two unconditional *branch* instructions and many conditional *branch* instructions, such as *branch on less than* and *branch on less than unsigned*.

The unconditional *branch* instructions allow you to specify a byte or word displacement. This allows you to branch to locations as far from the current location as 32,767 bytes in either direction. For control transfers to locations farther away, use the *jump* instruction.

▪ Cyclic Redundancy Check Instruction

The cyclic redundancy check (CRC) is an error detection method involving a division of the data stream by a CRC polynomial. In memory, the data stream is represented as a standard VAX string. Error detection is accomplished by computing the CRC polynomial at the source and again at the destination. The CRC is compared at each end. The CRC that is selected should minimize the number of undetected block errors of specific lengths.

The operands of the instruction are a string descriptor, a 16-longword table, and an initial CRC. The string descriptor is a standard VAX operand pair of the length of the string in bytes (up to 65,535) and the starting address of the string. The contents of the table are a function of the CRC polynomial to be used. It can be calculated from the polynomial by a variety of algorithms. The initial CRC figure is used to start the polynomial correctly. Typically, it has the value 0 or -1 but would be different if the data stream were represented by a sequence of noncontiguous strings.

The CRC instruction operates by scanning the string and for each byte of the data stream including it in the CRC being calculated. The byte is included by XORing it to the right eight bits of the CRC. Then the CRC is shifted right one bit, inserting zero on the left. The rightmost bit of the CRC (lost by the shift) is used to control the XORing of the CRC polynomial with the resultant CRC. If the bit is set, the polynomial is XORed with the CRC. Then the CRC is again shifted right and the polynomial is conditionally XORed with the result a total of eight times. Actual algorithms used can shift by one, two, or four bits at a time using the appropriate entries in a specially constructed table. The instruction produces a 32-bit CRC. For shorter polynomials, the result must be extracted from the 32-bit field. Data streams must be multiples of eight bits in length. If they are not, they must be right-adjusted in the string with leading 0 bits.

■ Decimal String Instructions

Decimal string instructions operate on packed decimal strings. They treat decimal strings as integers with the decimal point assumed immediately beyond the least significant digit of the string. If a string in which a result is to be stored is longer than the result, its most significant digits are filled with zeros.

Instructions are provided to convert between *packed decimal* and *trailing numeric* string (overpunched or zoned) and *leading separate numeric* string formats. Where necessary, a specific data type is identified. Where the phrase *decimal string* is used, it means any of the three previously mentioned data types. The instructions are

-
- *Add packed* (ADDP).
-
- *Arithmetic shift and rounded packed* (ASHP).
-
- *Compare packed* (CMPP).
-
- *Convert leading separate numeric string to packed decimal string* (CVTSP).
-
- *Convert longword integer to packed decimal string* (CVTLPL).
-
- *Convert packed decimal to leading separate string* (CVTPS).
-
- *Convert packed decimal string to a longword* (CVTPL).
-
- *Convert packed decimal string to a trailing numeric string* (CVTPT).
-
- *Convert trailing numeric to packed decimal string* (CVTTP).
-
- *Divide packed* (DIVP).
-
- *Move packed* (MOV).
-

-
- *Multiply packed* (MULP).
 - *Subtract packed* (SUBP).
-

A decimal string is specified by two operands.

-
- For decimal strings, the length is the number of digits in the string. The number of bytes in the string is a function of the length and the type of decimal string referenced.
 - The address of the lowest addressed byte of the string. This byte contains the most significant digit for *trailing numeric* and *packed decimal* strings. This byte contains a sign for *leading separate numeric* strings. The address is specified by a byte operand of address access type:
-

Each of the decimal string instructions uses general registers 0 through 3 or 0 through 5 to contain a control block that maintains updated addresses and state during the execution of the instruction. At completion, the registers containing addresses are available to the software to use as string specification operands for a subsequent instruction on the same decimal strings.

During the execution of the instructions, pending interrupt conditions are tested and, if any are found, the control block is updated. The *first part done* bit is set in the processor status longword, and the instruction is interrupted. After the interruption, the instruction resumes transparently. The format of the control block at completion is shown in Figure 6-2.

31		0
0		: R0
ADDRESS 1		: R1
0		: R2
ADDRESS 2		: R3
0		: R4
ADDRESS 3		: R5

Figure 6-2 ▪ Control Block after Instruction Execution

The fields ADDRESS 1, ADDRESS 2, and ADDRESS 3 (if required) contain the address of the byte containing the lowest addressed byte in the first, second, and third (if required) string operands, respectively.

Decimal overflow occurs if the destination string is too short to contain all the nonzero digits of the result. On overflow, the destination string is replaced by the correctly signed least significant digits of the result even if the result is -0. Note that neither the high nibble of an even length *packed decimal* string nor the sign byte of a *leading separate numeric* string is used to store result digits.

A zero result has a positive sign for all operations that complete without decimal overflow. However, when digits are lost because of overflow, a zero result receives the sign of the correct result.

A decimal string with a negative zero value is treated as identical to a decimal string with a positive zero value. For example, positive zero is equal to negative zero in a *compare* instruction. Similarly, when condition codes are affected on a negative zero result, they are affected as if the result were positive.

A reserved operand fault occurs if the length of a decimal string operand is outside the range of 0 through 31, or if an invalid sign or digit is encountered in a CVTSP or CVTTP instruction.

The result of any operation is *unpredictable* if any source decimal string operand contains invalid data. Except for CVTSP and CVTTP instructions, the decimal string instructions do not verify the validity of source operand data. If the destination operands overlap any source operands, the result of the operation is *unpredictable*. Destination strings, registers used by the instruction, and condition codes are *unpredictable* when a reserved operand fault occurs.

Packed decimal strings generated by the decimal string instructions always have the preferred sign representation—12 for positive and 13 for negative. An even length packed decimal string is always generated with a 0 digit in the high nibble of the first byte of the string. A packed decimal string contains an invalid nibble if

-
- A digit occurs in the sign position.
-
- A sign occurs in a digit position.
-
- A nonzero nibble occurs in the high-order nibble of the lowest addressed byte for an even length string.
-

The length of a packed decimal string can be zero. In this case, the value is zero (plus or minus) and one byte of storage is occupied. This byte must contain a 0 digit in the high nibble and the sign in the low nibble.

The length of a trailing numeric string can be zero. In this case, no storage is occupied by the string. If a destination operand is a zero-length trailing numeric string, the sign of the operation is lost. Memory access faults do not occur when a zero-length trailing numeric operand is specified because no memory reference occurs.

The length of a leading separate numeric string can be zero. In this case, one byte of storage is occupied by the sign. Memory is accessed when a zero-length operand is specified, and a reserved operand fault occurs if an invalid sign is detected. The value of a zero-length decimal string is zero.

▪ Edit Instruction

The edit instruction implements the common editing functions that occur in handling fixed format output. The instruction operates by converting a packed decimal string to a character string generating characters for the output. But the instruction can be used for other applications. When converting digits, options include leading zero fill; leading zero protection; insertion of floating sign, floating currency symbol, or special sign representations; and blanking an entire field when it is zero.

The operands to the EDITPC instruction are an input-packed, decimal-string descriptor, a pattern specification, and the starting address of the output string. The packed decimal descriptor comprises a standard VAX operand pair of the length of the decimal string of up to 31 digits and the starting address of the string. The pattern specification is the starting address of a *pattern operation editing sequence* that is interpreted in much the same way normal instructions are interpreted. Only the starting address of the output string is required because the pattern unambiguously defines the length.

While the EDITPC instruction is operating, it manipulates two character registers and the four condition codes. One character register contains the fill character. Normally, the character is an ASCII blank character. But the character may be changed to an asterisk (*) for check protection. The other character register contains the sign character. Initially, the character is either an ASCII blank or a minus sign depending upon the sign of the input. The sign register can be changed to allow other sign representations such as a plus or minus sign or plus/blank, and can be manipulated to output special notations such as CR for a credit (+) or DB for a debit (-). The sign register can also be changed to the currency sign in order to implement a floating currency sign.

After execution, the condition codes note the sign of the input, the presence of a nonzero source, an overflow condition, and the presence of significant digits. Condition code N is determined at the start of the instruction and is not changed except for correcting a negative zero input. The other condition codes are computed and updated as the instruction execution proceeds. When the EDITPC instruction terminates, registers 0 through 5 contain the conventional values after a decimal instruction.

■ Floating-point Instructions

Mathematically, a floating-point number may be defined as having the form: $\pm (2^K)f$ where K is an integer and f is a positive fraction. For a nonvanishing number, K and f are uniquely determined by imposing the condition: $1/2 < f < 1$.

The fraction factor (f) of the number is then said to be *binary normalized*. For the number 0, f must be assigned the value 0, and the value of K is indeterminate.

The VAX floating-point data formats are derived from this mathematical representation for floating-point numbers. Four types of floating-point data are provided; F__ floating numbers are 32 bits long, D__ floating and G__ floating numbers are 64 bits long, and H__ floating numbers are 128 bits long.

Because of the hidden bit, the fractional factor is not available to distinguish between zero and nonzero numbers whose fractional factor is exactly one half. Therefore VAX software reserves a sign-exponent field of zero for this purpose. Any positive floating-point number with biased exponent of zero is treated as if it were an exact zero by the floating-point instruction set. In particular, a floating-point operand, whose bits are all zero, is treated as zero. This is the format generated by all floating point instructions for which the result is zero.

A reserved operand is defined to be any bit pattern with a sign bit of 1 and a biased exponent of zero. On VAX machines, all floating-point instructions generate a fault if a reserved operand is encountered. Because a reserved operand has a biased exponent of 0, it can be internally generated only if overflow occurs.

An instruction is defined to be exact if its result extended on the right by an infinite sequence of zeros is identical to that of an infinite-precision calculation involving the same operands. The prior accuracy of the operands is thus ignored. For all arithmetic operations, except division, a 0 operand implies that the instruction is exact. The same statement holds for division if the 0 operand is the dividend. If it is the divisor, division is *undefined* and the instruction traps.

The add, subtract, multiply, and divide instructions, an overflow bit on the left, and two guard bits on the right are necessary and sufficient to guarantee return of a rounded result identical to the corresponding infinite-precision operation rounded to the specified word length. Thus with two guard bits, a rounded result has an error bound of one-half the least significant bit.

Note that an arithmetic result is exact if only no bits are lost in truncating the infinite-precision result to the data length to be stored. The first bit lost in truncating is called the *rounding* bit. The value of a rounded result is related to the truncated result as follows.

-
- If the rounding bit is 1, the rounded result is the truncated result incremented by a least significant bit.
-
- If the rounding bit is 0, the rounded and truncated results are identical.
-

Rounding may be implemented by adding a one to the rounding bit and propagating the carry if it occurs. Note that a renormalization may be required after rounding takes place. If this happens, the new rounding bit is zero so renormalization can happen once only. To summarize the relations among truncated, rounded, and true (infinite-precision) results.

-
- If a stored result is exact, then its rounded value = truncated value = true value.
-
- If a stored result is not exact, its magnitude is
 - always less than that of the true result for truncating.
 - always less than that of the true result for rounding if the rounding bit is 0.
 - greater than that of the true result for rounding if the rounding bit is 1.
-

To be consistent with the floating-point instruction set that faults on reserved operands, software-implemented, floating-point functions should verify that the input operands are not reserved. An easy way to do this is a *move* or *test* of the input operands.

In order to facilitate high-speed implementations of the floating-point instruction set, certain restrictions are placed on the addressing mode combinations usable within a single floating-point instruction. These combinations involve the logically inconsistent use of a value as both a floating-point operand and an address.

Specifically, if within the same instruction the contents of a specified register are used as an F__ floating point operand or part of a larger floating input operand and as an address in an addressing mode that modifies that register, the value of the floating-point operand is *unpredictable*. The operand specifier notation section describes the notation used for these instructions.

The VAX instruction set includes special floating-point instructions for modulus (range reduction) and polynomial calculation to aid in the implementation of mathematical functions, along with shift and rotate instructions.

The floating point instructions are

-
- *Add* (ADD).

 - *Clear* (CLR).

 - *Compare* (CMP).

 - *Convert* (CVT).

 - *Convert rounded* (CVTR).

 - *Divide* (DIV).

 - *Extended modulus* (EMOD).

 - *Move* (MOV).

 - *Move negated* (MNEG).

 - *Multiply* (MUL).

 - *Polynomial evaluation* (POLY).

 - *Subtract* (SUB).

 - *Test* (TST).

▪ Index Instruction

The index instruction calculates an index for an array of fixed-length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it aids index calculation optimization by removing invariant expressions.

▪ Integer Instructions

The integer optimizations include an instruction to write a longword onto the stack. Each integer data type includes operations that increment and decrement. The VAX instruction set includes special instructions to implement multiple precision integer arithmetic. A special variant of integer add instruction is an operation that adds a word under a memory interlock (for operating system counters in a multiprocessor system). The integer instructions are

-
- *Add aligned word under memory interlock* (ADAWI).

 - *Add with carry* (ADWC).

 - *Decrement* (DEC).

 - *Extended divide* (EDIV).

 - *Extended multiply* (EMUL).

 - *Increment* (INC).

 - *Push longword* (PUSHL).

 - *Subtract with carry* (SBWC).

▪ Logic Instructions

The logic computation instructions are for the three integer data types and are bit set (inclusive OR), bit clear (complement AND), and exclusive OR. The instructions are available in both two- and three-operand forms for each applicable data type. As input, the three-operand form takes the values of the first two operands and stores the result in the third operand.

The logical operations are simple *move*, *clear*, *arithmetic negate*, and *logical complement*. The *logical complement* operations are available only for the three-integer data types because these are the logical types. Both *negate* and *complement* include a *move*, rather than being restricted to altering an operand in place. VAX software has a large set of conversions covering almost all data type pairs. In addition, special conversions exist to round floating data to integer, and to extend unsigned integers to larger integers. The data comparison and testing instructions are compare, test against zero, and multiple bit testing. The logic instructions are

-
- *Arithmetic shift* (ASH).

 - *Bit clear* (BIC).

 - *Bit set* (BIS).

-
- *Bit test* (BIT).

 - *Clear* (CLR).

 - *Compare* (CMP).

 - *Convert* (CVT).

 - *Exclusive OR* (XOR).

 - *Move* (MOV).

 - *Move complemented* (MCOM).

 - *Move negated* (MNEG).

 - *Move zero-extended* (MOVZ).

 - *Rotate longword* (ROTL).

 - *Test* (TST).
-

▪ Multiple Register Instructions

Multiple register instructions save and restore several registers in one operation. The save area is on the stack. The PUSHHR instruction saves multiple registers by *pushing* them onto the stack. The POPR instruction restores multiple registers by *popping* them from the stack. A 16-bit mask operand specifies the list of registers. This mask is a normal read operand. The mask can be calculated or it can be an inline literal. When registers in the range R0 through R5 only are being saved or restored, the mask can be expressed as a short literal.

The software standard for calling and signaling requires that registers be saved in the call frame. With the exception of registers R0 and R1, any register manipulated by a PUSHHR or POPR instruction must appear in the procedure entry mask. The architecture also requires that any registers between R2 and R11 that are modified by the procedure to be saved in the call frame by setting up the appropriate entry mask. Registers R0 and R1 are used to return procedure status.

PUSHHR or POPR instructions should be used to save and restore only those registers specified in the procedure entry mask. If a procedure saves registers that are not noted in the entry mask and that procedure receives an exception, the procedure's caller's registers cannot be properly restored.

▪ Privileged Instructions

The privileged instructions give upward and downward mobility through the access modes, and provide a way to compare memory protection levels to the access mode privilege of callers. The instructions are

-
- *Change mode* (CHM).

 - *Extended function call* (XFC).

 - *Halt* (HALT).

 - *Load process context* (LDPCTX).

 - *Move from processor register* (MFPR).

 - *Move to processor register* (MTPR).

 - *Probe* (PROBE).

 - *Return from Exception or Interrupt* (REI).

 - *Save process context* (SVPCTX).

A *change mode* instruction is a special trap instruction that can be likened to an operating system service call instruction. User access-mode software can explicitly issue change mode instructions.

The *extended function* (XFC) instruction is used to request the services of non-standard microcode in the writeable control store or simulator software running in kernel mode. The request is controlled by the system control block.

The *halt* instruction is a privileged instruction that halts the processor only if it is running in kernel mode. If the instruction is issued when the processor is in any mode other than the kernel mode, a privileged instruction fault is issued.

When the operating system schedules a context switching operation, the context switching procedure uses the *save process context* (SVPCTX) and *load process context* (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

The *move to processor register* (MTPR) and *move from processor register* (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR instructions are privileged and can be issued only in kernel mode.

Probe instructions enable a procedure to compare the read (PROBER) and write (PROBEW) access protection of pages in memory to the privileges of the caller. The validation enables the operating system to provide services that execute in access modes to callers with less privileged access and yet prevent the caller from accessing protected memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the *return from exception or interrupt* (REI) instruction. The REI instruction is the only way the caller's access mode privilege can be decreased.

■ Procedure Call Instructions

Procedures are general purpose routines that use argument lists passed automatically by the processor and use only local variables for data storage. A procedure call instruction provides several services. It

-
- Saves all the registers that the procedure uses, and only those registers, before entering the procedure.
-
- Passes an argument list to a procedure.
-
- Maintains the *stack*, *frame*, and *argument* pointers.
-
- Sets the arithmetic trap enables to a specific state.
-

Three instructions are used to implement a standard procedure calling interface. Two instructions implement a procedure. The third instruction implements the matching *return* instruction. A *callg* instruction calls a procedure with the argument list actuals in an arbitrary location. The *calls* instruction calls a procedure with the argument list actuals on the stack. Upon return after a *calls* instruction, this list is automatically removed from the stack. Both *call* instructions specify the address of the entry point of the procedure being called. It is assumed to consist of a word called the entry mask followed by the procedure's instructions. The procedure terminates by executing a *return* instruction.

The entry mask specifies the subprocedure's register use and overflow enables. Figure 6-3 shows the entry mask.

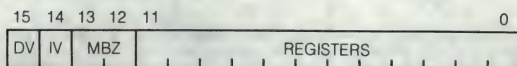


Figure 6-3 ■ Procedure Call Entry Mask

On *call*, the stack is aligned to a longword boundary and the *trap enable* bits in the processor status word are set to a known state to ensure consistent behavior of the called procedure. Integer overflow enable and decimal overflow enable are affected according to bits 14 and 15 of the entry mask, respectively. The *floating underflow enable* bit is cleared.

Registers R11 through R0 are saved on the stack and are restored by the *return* instruction. The procedure calling standard requires that all registers in the range R2 through R11 used in the procedure must appear in the mask. In addition, *call* instructions always preserve the program counter, stack pointer, frame pointer, and argument pointer. However, the stack pointer is not explicitly saved and differs after a *calls* instruction with arguments. Thus a procedure can be considered equivalent to a complex instruction that stores a value into R0 and R1, affects memory, and clears the condition codes. If the procedure has no function value, the contents of R0 and R1 are *unpredictable*.

In order to preserve the state, the procedure call instructions form a structure on the stack called a *call frame* or *stack frame*. This contains the saved registers and processor status word, the register save mask, and several control bits. The frame also includes a longword that the procedure call instructions clear. This is used to implement the condition handling facility. At the end of execution of the procedure call instruction, the frame pointer contains the address of the stack frame. The return instruction uses the contents of the frame pointer to find the stack frame and restore state. The condition handling facility assumes that frame pointer always points to the stack frame. See Figure 6-4 for the stack frame format.

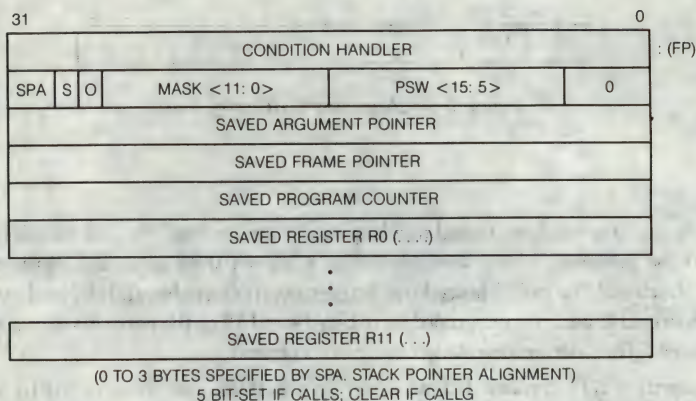


Figure 6-4 ■ Stack Frame Format

Note that the saved condition codes and the saved *trap enable* bits are cleared. The contents of bits 0 through 3 of the frame processor status word at the time *return* is executed becomes the condition codes resulting from the execution of the procedure.

■ Processor Status Longword Instructions

There are three instructions available to manipulate the processor status longword.

- *Bit clear processor status longword* (BICPSW) that clears a *trap enable* condition.
- *Bit set processor status longword* (BISPSW) that sets a *trap enable* condition.
- *Move from processor status longword* (MOVPSL) that obtains the processor status.

These are rather straightforward instructions and are not explained here. But the details on the instructions can be found in Chapter 9.

▪ Queue Instructions

A queue is a circular, doubly linked list whose entries are specified by their addresses. Each queue entry links to two others by way of a pair of longwords. The first or lower addressed longword is the forward link. It specifies the location of the succeeding entry. The second longword is the backward link. It specifies the location of the preceding entry. Two distinct types of queues are possible in VAX systems—absolute and self-relative. They are classified according to the type of links they use. An absolute link contains the absolute address of the entry to which it points. A self-relative link contains a displacement from the present queue entry.

Absolute Queue Instructions

An absolute queue is specified by a queue header that is identical to a pair of queue linkage longwords. The forward link of the header is the address of the entry called the *head of the queue*. The backward link of the header is the address of the entry termed the tail of the queue. The forward link of the tail points to the header.

Two general operations can be performed on queues—insertion and removal of entries. Generally, entries can be inserted or removed only at the head or tail of a queue.

The following figures illustrate some queue operations. An empty queue is specified by its header at address H as shown in Figure 6-5. If an entry at address B is inserted into an empty queue at either the head or tail, the queue is as shown in Figure 6-6. If an entry at address A is inserted at the head of the queue, the queue is as shown in Figure 6-7. Finally, if an entry at address C is inserted at the tail, the queue appears as shown in Figure 6-8. Following the steps above in reverse order gives the effect of removal at the tail and removal at the head.

If more than one process can perform operations on a queue simultaneously, insertions and removals should be done only at the head or tail of the queue. When just one process (or one process at a time) can perform operations on a queue, insertions and removals can be made at other locations. In the example above with the queue containing entries A, B, and C, the entry at address B can be removed as shown in Figure 6-9.

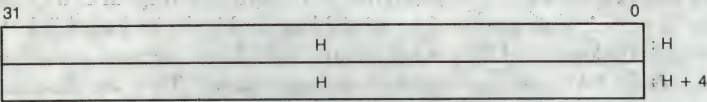


Figure 6-5 ■ Empty Absolute Queue

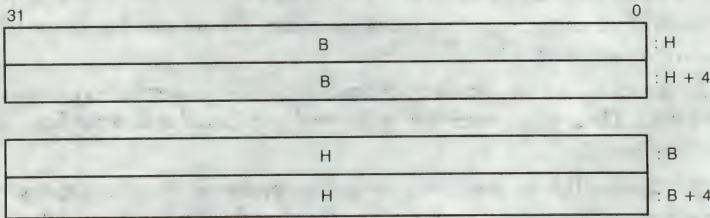


Figure 6-6 ■ Putting an Entry in an Empty Absolute Queue

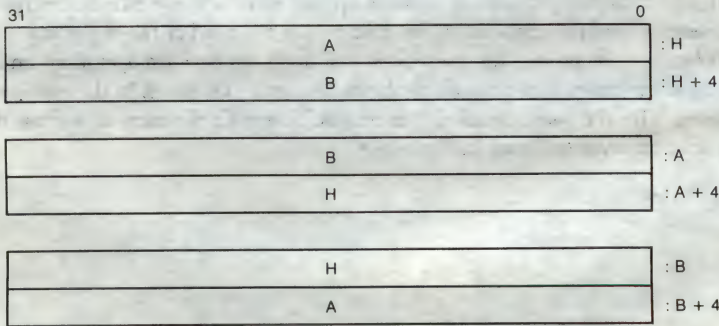


Figure 6-7 ■ Putting an Entry into the Head of an Absolute Queue

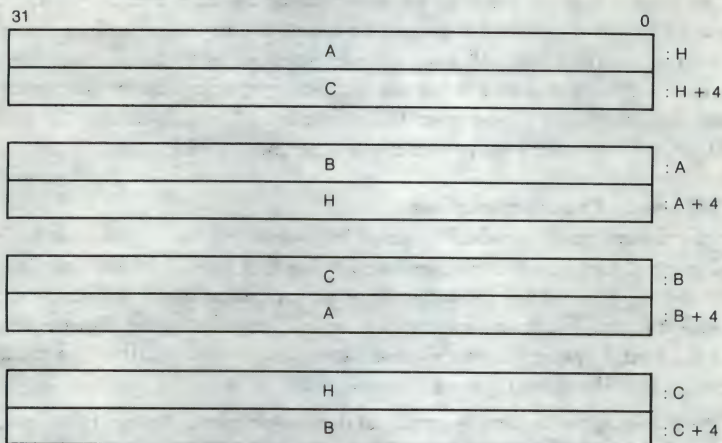


Figure 6-8 ■ Putting an Entry into the Tail of an Absolute Queue

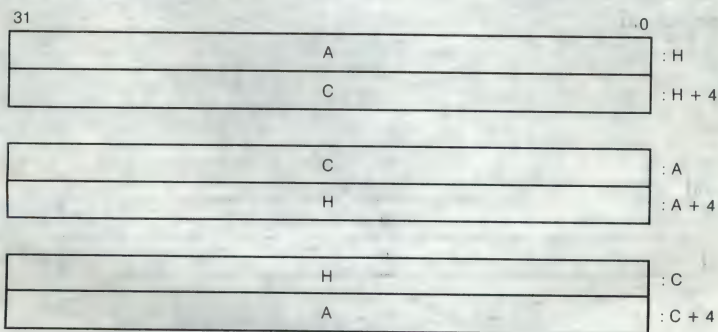


Figure 6-9 ■ Removing an Entry from an Absolute Queue

The reason for the restriction above is that operations at the head or tail are always valid because the queue header is always present. Operations elsewhere in the queue depend on specific entries being present and may become invalid if another process is concurrently performing operations on the queue.

Two instructions are provided for manipulating absolute queues—INSQUE and REMQUE. The INSQUE instruction inserts an entry specified by an entry operand into the queue, following the entry specified by the predecessor operand. The REMQUE instruction removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE instructions are implemented as noninterruptible instructions.

Self-relative Queue Instructions

Self-relative queues use displacements from queue entries as links. As with absolute queues, queue entries are linked by a pair of longwords. The first longword is the forward link displacement of the succeeding queue entry from the present entry. The second longword is the backward link—the displacement of the preceding queue from the present entry. A queue is specified by a queue header that also consists of two longword links.

The following shows some examples of queue operations. An empty queue is specified by its header at address H. Because the queue is empty, the self-relative links must be 0, as shown in Figure 6-10. If an entry at address B is inserted into an empty queue at either the head or tail, the queue is as shown in Figure 6-11. If an entry at address A is inserted at the head of the queue, the queue is as shown in Figure 6-12. Finally, if an entry at address C is inserted at the tail, the queue appears as shown in Figure 6-13. Following the steps above in reverse order yields the effect of removal at the tail and removal at the head.

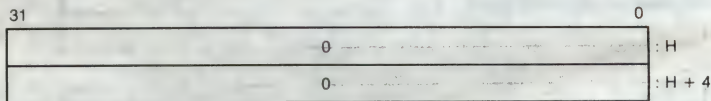


Figure 6-10 ■ Empty Self-relative Queue

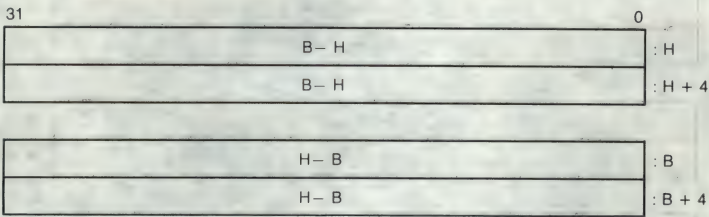


Figure 6-11 ■ Putting an Entry into an Empty Self-relative Queue

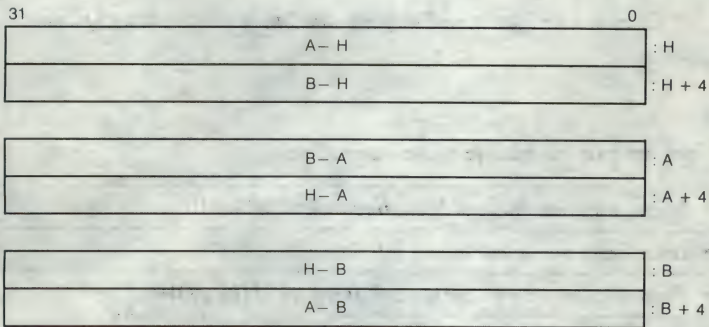


Figure 6-12 ■ Putting an Entry into the Head of a Self-relative Queue

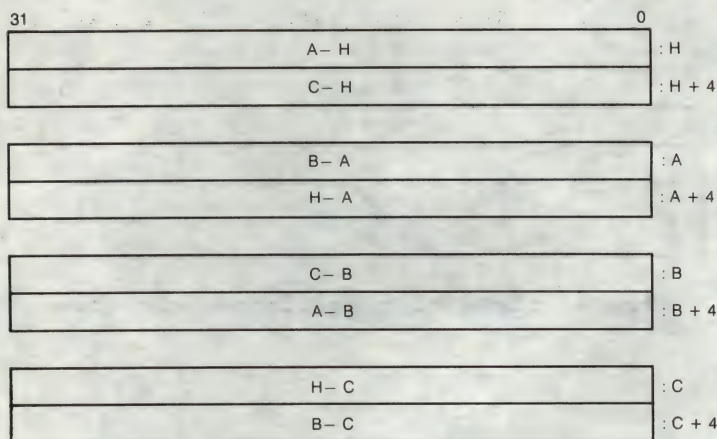


Figure 6-13 ■ Putting an Entry into the Tail of a Self-relative Queue

There are four self-relative queue instructions.

- *Insert entry into queue at head, interlocked (INSQHI).*
- *Insert entry into queue at tail, interlocked (INSQTI).*
- *Remove entry from queue at head, interlocked (REMQHI).*
- *Remove entry from queue at tail, interlocked (REMQTI).*

These operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared list without additional synchronization. Queue entries must be quadword aligned. A hardware-supported interlocked memory access mechanism is used to read the queue header. Bit 0 of the queue header is used as a secondary interlock and is set when the queue is being accessed.

If an interlocked queue instruction encounters the secondary interlock set, the instruction terminates after setting the condition codes to indicate failure to gain access to the queue. If the secondary interlock bit is not set, then the interlocked queue instruction sets the bit during its operation and clears the bit upon completion. This prevents other interlocked queue instructions from operating on the same queue.

▪ Variable Length Bit Field Instructions

Variable length bit field instructions are useful when dealing with data not in 8-bit increments; for example, 13 bits of data that do not start on a byte boundary. Such data could also be handled without these instructions but less efficiently because it requires additional shift and mask operations to get the bits into the proper form and to eliminate the nonrequired bits.

There are four variable length bit field instructions.

-
- *Compare field* (CMP).

 - *Extract field* (EXT).

 - *Find first* (FF).

 - *Insert field* (INSV).

The CMP instruction compares the field specified with a source operand. The EXT instruction causes the destination operand to be replaced by the specified sign-extended field. The FF instruction extracts a field specified by the start position, size, and base operand. The INSV instruction replaces a specified field with a base operand.

A variable bit field is 0 to 32 contiguous bits (contained in 1 to 5 bytes) that is arbitrarily located with respect to byte boundaries. The variable length bit field instructions have four operand specifiers. Three of these specifiers determine how to find the variable length field. The fourth designates where it is to be stored. The first three specifiers are the position operand, the size operand, and the base address.

The position operand is a signed longword operand that designates the number of bits away from the base address operand. If the variable length field is contained in a register, the position operand must have a value in the range 0 through 31 (if the size is nonzero) or a reserved operand fault occurs.

The size operand is a byte operand that specifies the length of the field. This operand must be in the range 0 through 32 or a reserved operand fault occurs. Normally, the size operand is a short literal if the field is fixed.

The base address is an address relative to which the position is used to locate the bit field. The base address is obtained from an address access type operand. Unlike other address access type operands, register mode may be designated in the specifier. In this case, the field is contained in register designated by the operand specifier.

Yakovlev's Landing Gear Problems

Yakovlev's Landing Gear Problems
The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem
is the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem is
the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem is
the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem is
the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem is
the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

The landing gear of the Yakovlev
fighter has been a source of
trouble for many years. The
main problem is the landing
gear itself, which is a
simple design but has many
problems. The main problem is
the landing gear itself, which
is a simple design but has many
problems. The main problem is
the landing gear itself, which is
a simple design but has many
problems.

Chapter 7 ■ Memory Management

Memory management is the control, allocation, and use of main memory for the VAX family of processors. VAX architecture is intended to support multiprogramming—the concurrent execution of a number of processes in a single computer system. A *process* can be defined for now as a single stream of machine instructions executed in sequence. Memory management includes both hardware and software. The hardware provides page mapping and protection, while the software provides the operating system image activator and pager.

Virtual address space is mapped into the physical address space by the processor's memory management logic. In addition, the memory management hardware supports *paging*. Paging is a technique that keeps in physical memory only those parts of a process's virtual memory that are in use. A VAX process exists in and operates on a memory space of 4,294,967,296 (2^{32}) bytes. Certain addresses and data are kept in the sixteen 32-bit general registers. A few processor state variables are kept in a special register called the *processor status longword*, or PSL. The combined set of information in memory, general registers, and PSL defines a process.

In a typical multiprogramming system, several processes may simultaneously reside in main memory. Memory protection is provided to ensure that one process does not affect other processes or the operating system. To improve software reliability further, memory access is controlled by the use of four privilege modes. They are kernel, executive, supervisor, and user. Kernel mode is the most privileged. User mode is the least privileged. Protection is specified at the individual page level. A page may be inaccessible, read only, or read/write for each of the four access modes. Any location accessible to a less privileged mode is also accessible to all more privileged modes. For each access mode, any location that may be written can also be read. While an image is being executed by the CPU, virtual addresses are generated. Before these addresses can be used to access instructions and data, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each 512-byte virtual page is located in main memory. The processor uses this mapping information in translating virtual addresses to physical addresses. Memory management provides both memory protection and memory mapping functions for VAX systems. This feature is designed to

-
- Provide a large address space for instructions and data.
 - Allow data structures up to one billion bytes.
-

-
- Provide convenient and efficient sharing of instructions and data.
 - Contribute to software reliability.
-

A virtual memory system is used to provide a large address space, while allowing programs to run on systems that have smaller memories. The operating system provides each process with a potential 4-billion-byte virtual address space.

▪ Virtual Memory

Half of the virtual address space is called *system space*. System space contains the operating system software and systemwide data. To facilitate interrupt handling and system service routines, system space is shared by all processes.

The other half of the virtual address space is separately defined for each process. It is called *process space* or per-process space. For consistency, we shall use the term *process space*. Process space is subdivided into P0 and P1 space. Program images and most of their data reside in P0 space. In P1 space, the system allocates space for stacks and process-specific data. Because P1 space is used for stacks, it is unique in that it is allocated from high addresses downward. Together, P0 and P1 space constitute a process's *working memory*. Except for special cases of sharing, each process has its own P0 and P1 spaces independent of others in the system. Figure 7-1 illustrates the address spaces of several processes in a multiprogramming system. Each process space is independent of the others. System space is shared by all.

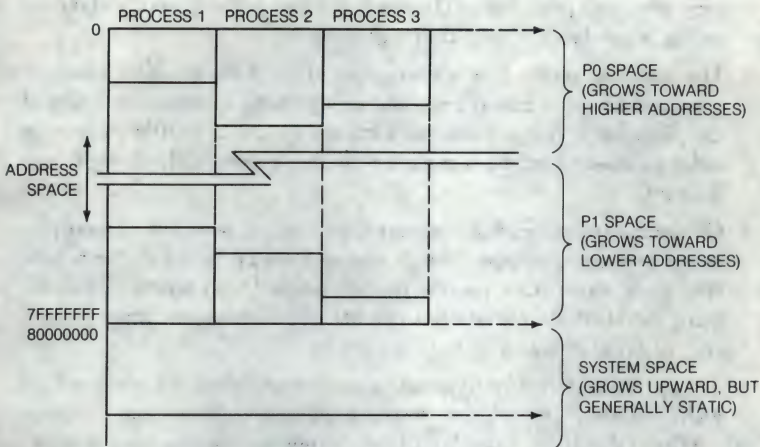


Figure 7-1 ■ Address Spaces in Process Context

Though the basic addressable unit in VAX machines is the 8-bit byte, larger units can be constructed by doubling byte sizes: a word is two bytes; a longword is four bytes; a quadword is eight bytes; and an octaword is sixteen bytes. These five are the units in which VAX memory stores data. But the processor sometimes interprets operands in other units; for example, half bytes (nibbles) for decimal digits, or variable-sized bit fields.

Generally, the memory system processes requests only for naturally aligned data. In other words, a byte can be obtained from any address. But a word can come only from an even address, and a longword can come only from an address that is a multiple of four, and so on. VAX processors convert an unaligned request into a sequence of requests that can be accepted by the memory. However, this conversion has a serious impact on performance. Data structures should be designed in such a way that the natural alignment of operands is preserved wherever possible.

The VAX memory management logic serves six principal purposes.

1. A number of processes may simultaneously occupy main memory. All processes can use process space addresses while referring independently to their own programs and data.

2. The operating system keeps selected parts of a process and its data in memory, bringing in other parts as needed without explicit intervention by the program. Large programs can be run in reduced memory space without re-coding or overlays visible to the programmer.
3. The operating system may scatter pieces of programs and data wherever space is available in memory without regard to the apparent contiguity of the program. It is never necessary for the system to shuffle memory in order to collect contiguous space for another process to be brought into memory.
4. Cooperating processes share memory in a controlled way. Two or more processes may communicate through shared memory, in which both have read/write access. One process may be granted read access to memory being modified by others; or, a number of processes may share a single copy of a read-only area.
5. The operating system limits access to memory according to a privilege hierarchy. Within any address space, privileged software can maintain databases that it can access but that code running in less privileged modes cannot.
6. The operating system may grant or inhibit access to control, status, and data registers in peripheral devices and their controllers. Since those registers are part of the physical address space, access to them is achieved by creation of a page table entry. The page frame number field of the page table entry selects the desired device or controller address in the I/O portion of the physical address space. References to the registers are then under control of the access control field of the page table entry. The same privilege mechanisms that control access to sensitive data in memory are used to control access to I/O devices.

For the purposes of memory management—specifically protection and translation of virtual to physical addresses—the unit of memory is the 512-byte page. Pages are always naturally aligned; that is, the address of the first byte of a page is a multiple of 512. Virtual addresses are 32 bits long, and are partitioned by the memory management logic as shown in Figure 7-2.

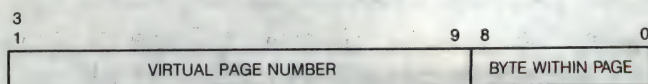


Figure 7-2 ■ Virtual Address Format

Field Extent: Bits 31:9

Field Name: Virtual Page Number

Function: The virtual page number field specifies the virtual page to be referenced. There are 8,388,608 pages in each virtual address space. Each page contains 512 bytes. When bit 31 is set (1), the address is in the system space. When bit 31 is clear (0), the address is in process space. Within the process space, bit 30 distinguishes between the program and control regions. When bit 30 is set (1), the control region is referenced. When bit 30 is clear (0), the program region is referenced.

Field Extent: Bits 8:0

Field Name: Byte Number

Function: The byte number field specifies the byte address within the page. A page contains 512 bytes.

The nine low-order bits select a byte within a page and are unchanged by the address translation process. The two high-order bits select the P0, P1, or system portion of the address space. The remaining 21 bits are used to obtain a longword called the page table entry (PTE) from the P0, P1, or system page table as appropriate. The page table entry format is described in detail later in this chapter. The PTE contains four pieces of information.

-
- *Protection code*—specifying which, if any, access modes are to be permitted read or write access to the page.
-
- *Page frame number*—identifying the 512-byte page of physical memory to be used on references to the virtual address.
-
- *Valid bit*—indicating that the page frame number is valid; that is, it identifies a page in memory rather than one in the swapping space on a disk.
-
- *Modification flag*—set by the processor whenever a write to the page occurs.
-

In concept, the process of obtaining a page table entry occurs on every memory reference. In practice, the processor maintains a translation buffer that is a special purpose cache of recently used page table entries. Most of the time, the translation buffer already contains the page table entries for the virtual addresses used by the program, and the processor need not go to memory to obtain them.

There is one page table entry for each existing page of the virtual address space. A length register associated with each region specifies how many pages exist in that region of the address space. The *system page table* (SPT) is allocated to contiguous pages in physical memory. The table contains page table entries for addresses greater than 80000000 (hexadecimal). Since the size of system space is relatively constant and can be determined at system startup time, allocating a fixed amount of physical memory to the system page table poses no problems.

Process space page tables change quite dynamically and can become very large. Because it would be awkward to require the operating system to keep the process page tables in contiguous areas of physical memory, VAX architecture defines structures called the *process space page tables*. The tables are identified as P0PT and P1PT and are to be allocated in contiguous areas of system space. Thus, the mapping for process space addresses involves two memory references: one to translate the process space address into a physical memory address, and the second to translate the system virtual address of the table containing the first translation. It is important to note that even if the translation buffer does not have the mapping for the process space address, it is likely to have that for the page table and can save one of the references.

■ Virtual Address Space

The virtual address space is divided into two address spaces of equal size; one for the processes, the other for the system. The system address space is the same for all processes. The operating system resides in the lower half of the system address space. The operating system is implemented as a series of callable procedures. This arrangement makes the system code available to all other system and user codes using a *call* instruction. The upper half of the system space is reserved for future use. Process address space is separate for each process. However, several processes may have access to the same page thus providing controlled sharing. A virtual address is a 32-bit unsigned integer specifying a byte location in the address space. The address space seen by the programmer is a linear array of over 4 billion bytes. The space is divided into a collection of 512-byte units called *pages*. The page is the basic unit of both relocation and protection.

Virtual address space cannot be contained in currently manufactured main memory. Memory management maps the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating system controls the memory management tables that map virtual addresses into main memory addresses. Parts of the virtual address space that are not in use are copied or *swapped* to auxiliary memory. When those parts are needed, they are brought back into the virtual address space. See Figure 7-3 for a diagram of virtual address space.

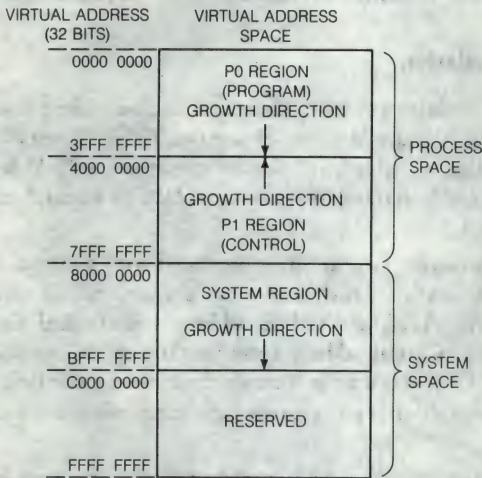


Figure 7-3 ■ Virtual Address Space

The figure shows that virtual address space is divided into two major areas—process space and system space. Each process has a separate address translation map for process space, so the process spaces of all processes are noncontiguous.

The address map for process space is context-switched when the process running on the system is changed. Process space is further divided into two regions named P0 and P1. These regions are described in detail later in this chapter.

The other half of virtual address space is called *system space*. All processes use the same address translation map for system space. System space is shared by all processes. The address map for system space is not context-switched. In a shared-memory multiprocessor configuration, changing any of the address mapping information for system space requires that all processors execute an *MTPR. xxx, #TBIS* instruction.

Access to each of the three regions is controlled by a length register. The length registers are P0LR, P1LR, and SLR. Register P0LR controls access to region P0. Register P1LR controls access to region P1. Register SLR controls access to the system space of the virtual address space. Within the limits set by the length registers, the access is controlled by a page table that specifies the validity, access requirements, and location of each page in the region.

■ Address Translation

The action of translating a virtual address to a physical address is governed by the setting of the Memory Mapping Enable (MME) bit. When MME is reset (0), page protection is disabled. This feature is reserved for Field Service. This section describes address translation when the MME bit is set (1) and page protection is enabled.

The address translation mechanism is presented with a virtual address, an intended access (read or write), and a mode against which to check that access. If the access is allowed and the address is not faulted, the output of this routine is a physical address corresponding to the specified virtual address. The mode that is used is normally the current mode field of the processor status longword. But process page table entry references use the kernel mode.

If the operation to be performed is a read operation, the intended access is read access. If the operation to be performed is a write operation, the intended access is write access. If the operation to be performed is a modify (that is, a read followed by a write operation), the intended access for the read portion is specified as a write access. If an operand is not an address operand, no reference is made.

Page Table Entry

All virtual addresses are translated to physical addresses by means of a page table entry (PTE). See Figure 7-4 for a graphic description of the page table entry.

Field Extent: Bit 25

Field Name: Must Be Zero

Function: This bit is reserved and must be zero.

Field Extent: Bits 24:23

Field Name: Owner (OWN) bits

Function: These bits are reserved for system software use. The VAX/VMS operating system uses these system bits as the access mode of the owner of the page; that is, the mode allowed to alter the page. The field is not examined or altered by hardware.

Field Extent: Bits 22:21

Field Name: Operating System Software

Function: These bits are reserved for Operating System Software.

Field Extent: Bits 20:0

Field Name: Page Frame Number (PFN)

Function: The upper 21 bits of the physical address of the base of the page. The field is used by hardware only if the *valid* bit is set.

The operating system software uses combinations of software bits to implement its page management data structures and functions. Some of the functions implemented are initialize pages with zeros, copy on reference, page sharing, and transitions between active and swapped-out states. VAX/VMS software encodes these functions in page table entries whose *valid* bit is reset (0) and processes them whenever a page fault occurs.

Page Table Entry for I/O Devices

Some I/O devices use memory management to translate addresses. These devices use a page table entry format that is an extension of the page table entry used by the CPU. For hardware, the extended page table entry implements some functions that the CPU implements with software. Three page table entry bits are used in four combinations to identify a valid page frame number, a global page table index, and an I/O abort. The page table bits are 22, 26, and 31. The page frame number is valid if bit 31 is set (1) or if bits 22, 26, and 31 are reset (0). When bit 22 is set (1) and bits 26 and 31 are reset (0), the page frame number field is a global page table index (GPTX). The I/O device has a global page table base register that is loaded with a system virtual address.

The I/O device calculates the system virtual address of a second page table entry. The second page table entry must yield a valid page frame number and the three bits must indicate a valid page frame number. If either of these requirements is not met, the result is *undefined*. The protection field always comes from the first PTE. Some I/O devices may examine and check the protection field or modify the M bit—this is device dependent. Devices that use the protection field and M bit do so in the same manner as does the CPU.

I/O devices that perform memory mapping use the same SPT as the CPU. But the devices have their own copies of the system base register and system space length register. Buffer addresses are described in terms of a system virtual address of the PTE for the first buffer page and a byte offset within that page. In addition, the I/O devices use a global page table in memory and an I/O hardware global page table base register (GBR) which must be loaded by software.

Changing Page Table Entries

Page table entries are changed by the operating system as part of its memory management functions. For example, the operating system sets and resets the *valid* bit and changes page frame numbers as pages are swapped.

The software must guarantee that each PTE is consistent within itself. Changing a PTE one field at a time may cause incorrect system operation. For example, the *valid* bit could be set for one instruction while the page frame number is for another instruction. Then, an interrupt routine could occur between the two instructions that would use an address mapped to this inconsistent PTE. This problem can be avoided by simultaneously changing all the fields in PTE. Simply build the new PTE in a register and move that PTE into the page table with one instruction (MOVL).

Multiprocessors complicate the matter. One processor can reference a page table that is being modified by another processor. The PTEs must be consistent. In order to guarantee this, first note that PTEs are longword-aligned longwords. Because of this, two requirements must be met. First, whenever the software modifies a PTE in more than one byte, the software must use a longword, longword-aligned, write-destination instruction (such as MOVL). Second, the hardware must guarantee that an incomplete longword-aligned longword write operation cannot be read or overwritten.

System Space Address Translation

Figure 7-5 graphically describes the system space address format. The figure shows a virtual page number field with bits 31 and 30 set to a value of 2.

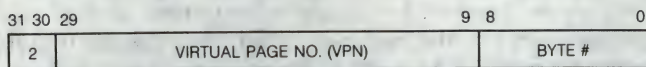


Figure 7-5 ■ System Space Address Format

The system virtual address space is defined by the system page table, which is a vector of page table entries. The physical base address of the system page table is in the system base register. The size of the system page table in long-words (number of page table entries) is in the system length register. The page table entry addressed by the system base register maps the first page of system space; that is, virtual byte address 80000000 (hexadecimal).

The virtual page number field is bits 9 through 29 of the virtual address. Thus, there could be as many as 2,097,152 (2^{21}) pages in the system region. Typically, the value is in the range of a few hundred to a few thousand system pages. A 22-bit field is required to express the values 0 through 2,097,152 inclusive. During a bootstrap procedure, the contents of both registers are *unpredictable*. The translation from system space virtual address to physical address is shown in Figure 7-6.

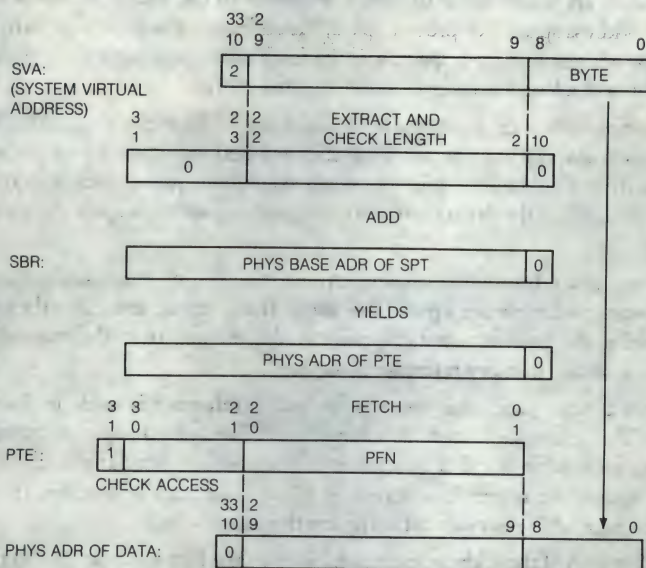


Figure 7-6 • System Space Address Translation

Process Space Address Translation

The process virtual address space is divided into two separately mapped regions according to the setting of bit 30 in the process virtual address. If bit 30 is reset (0), the P0 region of the address space is selected. If bit 30 is set (1), the P1 region is selected.

The P0 region of the address space defines a contiguous area starting at the smallest address in the process virtual space and moving toward the larger addresses. The P0 region is used typically for program images, and the region grows dynamically.

In contrast, the P1 region of the address space defines a contiguous area that starts at the largest address in the process virtual space and moves toward the smaller addresses. The P1 region is typically used for system-maintained process context. It may grow dynamically for the user stack.

Both regions of the process virtual space are described by page tables. The two page tables are addressed with virtual addresses in the system region of the virtual address space. For process space, the address of the page table entry is a virtual address in system space, and the fetch of the page table entry is simply a fetch of a longword using a system virtual address.

Process page tables are addressed in virtual space. If the tables were addressed in physical space, a process page table that required more than a page of page table entries would also require physically contiguous pages. Such a requirement would make the dynamic allocation of process page table space more complex.

A process space translation causing a translation buffer miss usually causes one memory reference for a page table entry. If the virtual address of the page containing the process page table entry is also missing from the translation buffer, a second memory reference is required.

When a process page table entry is fetched, a reference is made to system space. This reference is made as a *kernel read*. The system page containing a process page table is either accessible or inaccessible. Similarly, a check is made against the system length register (SLR). The fetch of an entry from a process page table can cause access or length violation faults.

The P0 region of the address space is mapped by the P0 page table (P0PT) that is defined by the P0 Base Register (P0BR) and the P0 Length Register (POLR). The base register contains a virtual address in the system half of virtual address space that is the base address of the P0 page table. The length register contains the size of the P0 page table in longwords; that is, the number of page table entries. The page table entry addressed by the P0 base register maps the first page of the P0 region of the virtual address space (virtual byte address zero). The page table entries in the P0 page table contain the mapping information themselves; or point to the mapping information in the global page table if the page table entry is in the global page table index format.

The virtual page number is bits 9 through 29 of the virtual address. This means there could be as many as 2,097,152 (2^{21}) pages in the P0 region. A 22-bit field is needed to express the values 0 through 2,097,152 inclusive. Bits 24 through 26 of register POLR are ignored on the *move to processor register* (MTPR) instruction and are read back as zero on the *move from processor register* (MFPR) instruction. During bootstrap procedures, the contents of both registers are *unpredictable*. An attempt to load register P0BR with a value less than 2,147,483,648 (2^{31}) or greater than 3,221,225,468 ($((2^{31}) + (2^{30})) - 4$) causes a reserved operand fault. The translation from P0 virtual address to physical address is shown in Figure 7-7.

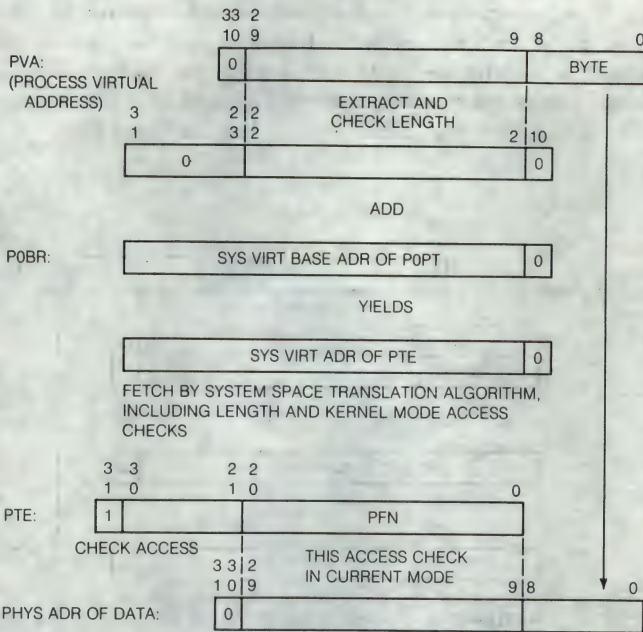


Figure 7-7 • P0 Region Address Translation

The P1 region of the address space is mapped by the P1 page table (P1PT) that is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space moves from higher to lower addresses and because a consistent hardware interpretation of the base and length registers is important, registers P1BR and P1LR describe that portion of P1 space that is inaccessible. The base register contains a virtual address of what would be the page table entry for the first page P1-virtual byte address 40000000 (hexadecimal). The length register contains the number of nonexistent page table entries. An address in the base register is not necessarily an address in system space; but an address of a page table entry must be in system space. The page table entries in the P1 page table contain the mapping information or point to the mapping information in the global page table entry if the page table entry is in global page table index format.

Bit 31 of the length register is ignored by a *move to processor register* instruction and is read as zero by a *move from processor register* instruction. During bootstrap procedures, the contents of both registers are *unpredictable*. Attempting to load register P1BR with a value less than 2,139,095,040 (7F800000 (hexadecimal)) causes a reserved operand fault. The translation from P1 virtual address to physical address is shown in Figure 7-8.

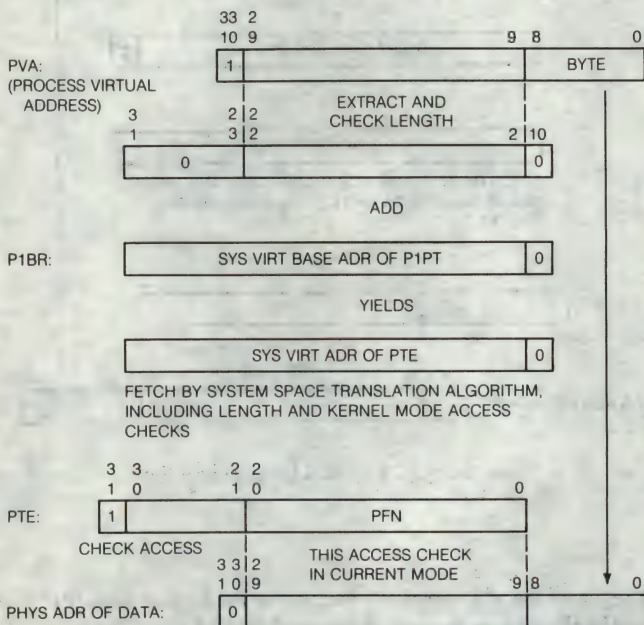


Figure 7-8 ■ P1 Region Address Translation

■ Access Control

Access control is the process of screening page access requests and verifying that the requester is authorized access to that page. Every page is assigned a protection code. That code specifies for each mode whether or not read or write references are allowed. Also, each address is checked to ensure that it resides in the virtual address space.

There are four access modes. The modes are listed in Table 7-1 in the order of most to least privileged. The mode at which the processor is currently running is stored in the current mode field of the processor status longword.

Table 7-1 • Processor Access Modes

Access Mode Code	Access Mode Name
0	Kernel
1	Executive
2	Supervisor
3	User

Page protection is assigned according to its use, *not* its location in the virtual address space. Although the system space is shared, a program may be prevented from modifying or accessing portions of the system space. A program may also be prevented from accessing or modifying portions of process space. For example, in system space, scheduling queues are highly protected, but library routines may be executed by any privilege code. Also, in process space, process accounting information may be highly protected, while normal user code may be executed by any privilege code.

Each page is assigned a protection code describing the accessibility of the page for each mode. The protection codes allow a choice of protection for each access level within the following limits.

- Each level's access can be read or write, read only, or no access.
- Whichever level has read access, all more privileged levels also have read access.
- Whichever level has write access, all more privileged levels also have write access.

This scheme results in 15 possible protection codes. The protection code is encoded in a 4-bit field in the page table entry. See Table 7-2 for a list of the codes.

Table 7-2 ■ Page Table Entry Protection Codes

Protection Code			Access Mode†				Comments
Decimal	Binary	Mnemonic*	K	E	S	U	
0	0000	NA	NO	NO	NO	NO	No access
1	0001		UN	UN	UN	UN	Reserved
2	0010	KW	RW	NO	NO	NO	Kernel write
3	0011	KR	R	NO	NO	NO	Kernel read
4	0100	UW	RW	RW	RW	RW	All access
5	0101	EW	RW	RW	NO	NO	
6	0110	ERKW	RW	R	NO	NO	
7	0111	ER	R	R	NO	NO	
8	1000	SW	RW	RW	RW	NO	
9	1001	SREW	RW	RW	R	NO	
10	1010	SRKW	RW	R	R	NO	
11	1011	SR	R	R	R	NO	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

* Software symbols are defined using PTE\$K as a prefix to the mnemonics listed above. For example, the software protection symbol PTE\$KUR means that that software can be read by anyone with user access privileges and those with higher privileges. A software protection symbol of PTE\$KER means that that software can be read by anyone with kernel or supervisor access privileges. No one is allowed write access to that software.

† There are four access modes—K for Kernel Access Mode, E for Executive Access Mode, S for Supervisor Access Mode, and U for User Access Mode. Within these access modes, there are certain functions that may be performed—R indicates read access only, RW indicates read and/or write access, NO indicates no access, and UN indicates *unpredictable* results if access is attempted.

Every valid virtual address must reside in one of the addressing regions and the associated length registers. The algorithm for making these checks is a limit check. The notation for this check is shown in Example 7-1.

Example 7-1 • Valid Virtual Address Checking Algorithm

```

case VAddr (31:30)
  set
  [0]: if ZEXT (VAddr(29:9)) GEQU P0LR      !P0 region
        then {length violation}
  [1]: if ZEXT (iVAddr(29:9)) LSSU P1LR      !P1 region
        then {length violation};
  [2]: if ZEXT (VAddr(29:9)) GEQU SLR        !S region
        then {length violation};
  [3]:                                     !reserved region
        {length violation};
tes;

```

An access control fault occurs if the current mode of the processor status longword and the page protection fields indicate the access is illegal, or if the address causes a length violation. If an access is made across a page boundary, the order in which the pages are accessed is *unpredictable*. However, for a given page, access control violation always takes precedence over translation not valid.

▪ Controlling Memory Management

There are three additional privileged registers used to control the memory management hardware. One register is used to enable and disable memory management. The other two are used to clear the hardware address translation buffer when a page table entry is changed.

The action of translating a virtual address to a physical address is governed by the setting of the *memory mapping enable* bit of the map enable register. The map enable register (MAPEN) contains a value of 0 or 1 depending upon memory management. If memory management is disabled, the value is 0. If memory management is enabled, the value is 1. During bootstrap procedures, this register is initialized to zero.

When memory management is disabled, virtual addresses are mapped to physical addresses by ignoring their high-order bits. Access is allowed in all modes, and the modify bit is not maintained.

To read the register, use the *move from processor register* (MFPR) with the source operand specified as #56. To write to the register, use the *move to processor register* (MTPR) instruction using #56 as the destination operand.

In order to reduce repeated address translations, the hardware includes a translation buffer that records virtual address translations and page status. Whenever the process context is loaded by the *load process context* (LDPCTX) instruction, the translation buffer is automatically updated. That is, the process virtual address translations are invalidated. Whenever a page table entry for the system or current process region is changed other than to set the page table entry *valid* bit, the software must notify the translation buffer of this change. This is done by moving an address within the corresponding page into the *translation buffer invalidate single* (TBIS) register.

Additionally, when the software changes a system page table entry that maps any part of the current process page table, all process pages so mapped must be invalidated in the translation buffer. They may be invalidated by moving an address within each such page into the TBIS register. They may also be invalidated by clearing the entire translation buffer. This is done by moving zero to the *translation buffer invalidate all* register with a *move to processor register* instruction.

The translation buffer must not store invalid page table entries. Software is not required to invalidate translation buffer entries when making changes for page table entries that are already invalid. Whenever the location or size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared by moving 0 into the *translation buffer invalidate all* (TBIA) register. Before enabling memory management, the translation buffer must be cleared by moving 0 into the TBIA register with the *move to processor register* instruction.

Whenever the *memory management enable* bit is zero, the contents of the translation buffer are *unpredictable*. Therefore, the entire translation buffer must be cleared before enabling memory management.

■ Faults and Parameters

Two types of faults are associated with memory mapping and protection. A *translation not valid* fault is taken when a read or write reference is attempted through an invalid page table entry. An *access control violation* fault is taken when the protection field of the page table entry indicates that the intended access is illegal.

Note that these two faults have distinct vectors in the system control block. If both *access control violation* and *translation not valid* faults occur, the *access control violation* fault takes precedence.

An *access control violation* fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a length violation is essentially the same as referencing a page table entry that specifies *no access*. To avoid repeating the length check, a *length violation* is stored in the *fault parameter word*. The *fault parameter word* format is shown in Figure 7-9.

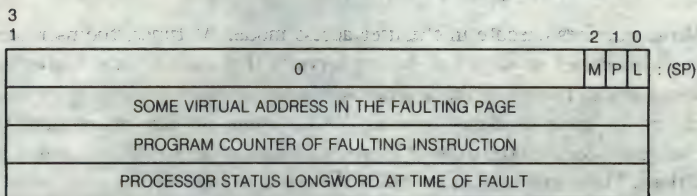


Figure 7-9 ■ Fault Parameter Word Format

The same parameters are stored for both types of faults. The first parameter pushed on the kernel stack after the processor status longword and program counter is the initial virtual address that caused the fault. A process space reference can result in a system space virtual reference for the page table entry.

If the page table entry reference faults, the process virtual address is saved. In addition, a bit is stored in the fault parameter word indicating that the fault occurred in the process page table entry reference. The second parameter pushed on the kernel stack contains the information listed below.

Field Extent: Bit 2

Field Name: Write or Modify Intent

Function: This bit is set (1) to indicate that the program's intended access is a write or modify. This bit is reset (0) if the program's intended access is a read.

Field Extent: Bit 1

Field Name: Page Table Entry Reference

Function: This bit is set (1) to indicate that the fault occurred during the reference to the process page table associated with the virtual address. This bit is set on either length or protection faults.

Field Extent: Bit 0

Field Name: Length Violation

Function: This bit is set (1) to indicate that an access control violation was the result of a length violation rather than a protection violation. This bit is reset (0) for a *translation not valid* fault.

■ Accessing Privileged System Services

Most processes execute in the user access mode. At times, the user access mode processes need to use system services that execute at a higher-level access mode. These user mode processes are not allowed access to other modes except for these necessary services. VAX systems provide instructions that change an access mode to one of greater privilege under strictly controlled conditions. These instructions are called the privilege instructions.

The privilege instructions change a process's mode to a more privileged mode and transfer control to a service dispatcher for the new mode. The instructions provide the only mechanism for less privileged code to call more privileged code. When the mode transition takes place, the previous mode is saved in the previous mode field of the processor status longword. This allows the more privileged code to determine the privilege of its caller.

The instructions give upward and downward mobility through the access modes, and provide a way to compare memory protection levels to the access mode privilege of callers. The instructions are *change mode* (CHM), *probe*, *return from exception or interrupt* (REI), *save process context* (SVPCTX), *load process context* (LDPCTX), *move to processor register* (MTPR), and *move from processor register* (MFPR).

User mode software can access privileged services by calling operating system service procedures with a *call* instruction. The operating system's service dispatcher issues an appropriate *change mode* instruction before entering the procedure. A *change mode* instruction allows access mode transitions to take place from one mode to the same or more privileged access mode (upward) only. When such a mode transition takes place, the previous mode is saved in the previous mode field of the processor status longword. This action allows the more privileged code to determine the access privilege of its caller.

A *change mode* instruction is a special trap instruction that can be likened to an operating system service call instruction. User access-mode software can explicitly issue change mode instructions. Because the operating system receives the trap, nonprivileged users cannot write software to execute in any of the privileged access modes. User mode software can include a condition handler for change mode to user traps. This instruction provides general purpose services for user access-mode software. Before software with a change mode instruction can be executed, the user's privilege must be changed in the system authorize database (SYSUAF.DAT). The privilege is changed for one program and is not a global change for the user.

For service procedures written to execute in privileged access modes, the processor provides address access privilege validation instructions called *probe* instructions. *Probe* instructions enable a procedure to compare the read (PROBER) and write (PROBEW) access protection of pages in memory to the privileges of the caller. The validation enables the operating system to provide services that execute in access modes to callers with less privileged access and yet prevent the caller from accessing protected memory.

When the operating system schedules a context switching operation, the context switching procedure uses the *save process context* (SVPCTX) and *load process context* (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers include not only those that identify the executing process but also the memory management and other registers such as the console and clock control registers. The *move to processor register* (MTPR) and *move from processor register* (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR instructions are privileged and can be issued only in kernel mode.

The operating system's privileged service procedures and interrupt and exception service routines exit using the *return from exception or interrupt* (REI) instruction. The REI instruction is the only way the caller's access mode privilege can be decreased. Like the procedure and subroutine return instructions, the REI instruction restores the program counter and the processor state to the values that were stored there before the change mode trap interruption. This procedure ensures that the process resumes execution at the point where it was interrupted.

REI instructions perform special services that normal return instructions do not. For example, REI instructions inspect the asynchronous system trap queue for the executing process. If an asynchronous system trap was queued for the process while the interrupt or exception service routine was executing, the REI instruction ensures that the process receives them. Also, REI instructions check the mode to which control is being returned. That mode must be the same as or a less privileged mode than the one in which the exception or interrupt occurred. The REI instruction is available to all software including user-written trap handling routines. There is a restriction; a program cannot increase its privilege by altering the processor state to be restored.

Events within the system sometimes need software outside the flow of control. In these cases, the processor changes the flow of control from that indicated in the executing process. Some such events are relevant to the current process and normally invoke software within the context of that process. The notification of these events is called an *exception*.

Other events are relevant to other processes or to the system as a whole and are serviced in a systemwide context. The notification process for these events is called an *interrupt*. The systemwide context is described as *executing on the interrupt stack*. Some interrupts require high-priority service, while others must be synchronized with independent events. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any moment. The priority assigned to an interrupt is called its *interrupt priority level* (IPL).

■ Event Handling

Exceptions are handled by the operating system. Usually they are *reflected* to the originating mode as a signal. In general, the exception is described by a vector that is a list of longwords. The first longword contains a count of other longwords in the vector. The second longword identifies which exception occurred. The remaining longwords are the stack parameters, the program counter, and the processor status longword. There are three kinds of exceptions — *aborts*, *faults*, and *traps*.

An *abort* is a condition that occurs when an instruction leaves the value of the registers and memory in an *unpredictable* condition and the instruction cannot be correctly restarted, completed, simulated, or undone. After an instruction aborts, the program counter addresses the opcode of the aborted instruction.

The following events produce *unpredictable* results.

-
- Destination operands including implied operands such as the top of the stack in a JSB instruction.
-
- Registers modified by an operand specifier evaluation including specifiers for implied operands.
-
- The *modify* bit of the page table entry in those entries that map destination operands if the operands could have been but were not written, and the modify bit was clear before the instruction.
-

-
- The *first part done* bit of the processor status longword.
 - The *trace pending* bit of the processor status longword.
-

If not noted in the description of the abort exception, the rest of the processor status longword, other registers, and memory are not affected.

A *fault* exception is a condition that occurs during an instruction. Faults leave the registers and memory in a consistent state. When the fault condition is corrected and the instruction is restarted, the execution yields correct results. Note that faults do not always leave everything as it was prior to the fault instruction. Faults restore only enough to allow restarting. Thus the state of a faulted process may not be the same as that of an interrupted process if both occurred at the same point.

A *trap* exception is a condition that occurs at the end of the instruction that caused the exception. Therefore, the program counter saved on the stack is the address of the next instruction that would normally have been executed. Any software can enable and disable some of the trap conditions with a single instruction. For example, refer to the descriptions of the *bit set processor status word* (BISPSW) and *bit clear processor status word* (BICPSW) instructions.

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the current interrupt priority level does the processor raise the level and service of the interrupt request. The interrupt service routine is entered at the level of the interrupt request and usually does not change the set by the processor.

Interrupt requests come from devices, controllers, other processors (in customer-designed systems), or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor. But note that the priority level of one processor does not affect the priority level of the other processors. This is done to prevent the interrupt priority levels from being used to synchronize access to shared resources in multiprocessor systems. Special software action is required to stop other processors in your multiprocessor system.

Most service routines for software-generated exceptions execute at interrupt priority level 0. However, if a serious system failure occurs, the processor raises the interrupt priority level to the highest level to prevent interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions. However, nested exceptions may occur in the the following faults—an access control violation, reserved operand, or reserved addressing mode.

Interrupt Priority Levels

The processor has 31 interrupt priority levels (IPLs) divided into 15 software levels (numbered 1 through F (hexadecimal)) and 16 hardware levels (10 through 1F (hexadecimal)). User applications, system calls, and system services run at process level (IPL0). Higher numbered IPLs have higher priority. Any request with an IPL higher than the processor's IPL causes an immediate interrupt. But requests with a lower or equal IPL are deferred.

Interrupt levels 1 through F (hexadecimal) exist entirely for use by software. No hardware device can request interrupts on those levels but software can force an interrupt. The interrupt is forced by executing a *move to processor register* instruction using the software interrupt request register as the destination. After a software interrupt request is made, the request is cleared by hardware when the interrupt is taken. Interrupt levels 10 through 17 (hexadecimal) are for use by devices and controllers, including UNIBUS devices. Interrupt levels 18 through 1F (hexadecimal) are used by urgent conditions including—the interval clock, serious errors, and *powerfail*.

Two of the software interrupt priorities are reserved for process structure software. They are IPL2 and IPL3. IPL2 is the AST delivery interrupt. It is triggered by a *return from exception or interrupt* instruction that detects $PSL < CUR MOD > GEQU ASTLV L$. IPL2 indicates that a pending AST for the executing process can now be delivered.

IPL3 is the process scheduling interrupt. It is triggered by software to allow the process running at IPL3 to cause the executing process to be blocked and the highest priority executable process to be scheduled.

Exceptions and Interrupts

Exceptions and interrupts are similar. When either is initiated, both the processor status longword (PSL) and the program counter are put on a stack. However, there are seven important differences between exceptions and interrupts.

-
- An exception is caused by an executing instruction. An interrupt is caused by the computing system and is usually independent of an instruction.
-
- Usually, an exception is serviced in the context of the process that produced that exception. An interrupt is serviced independently of the current process.
-
- Generally, the interrupt priority level of the processor is not changed when the processor initiates an exception. The interrupt priority level is always raised when an interrupt is initiated.
-

-
- Normally, exception service routines execute on a process stack. Interrupt service routines normally execute on a processor stack. However, a *machine check* always executes on the interrupt stack pointer.
-
- Enabled exceptions are initiated immediately, independent of the processor interrupt priority level. Interrupts are delayed until the processor interrupt priority level drops below the level of the requesting interrupt.
-
- Most exceptions cannot be disabled. If an exception-causing event should occur while that exception is disabled, no exception is initiated even when that event is subsequently enabled. This includes *overflow* exceptions. If an interrupt is disabled and an initiating event occurs, the event initiates an interrupt when subsequently enabled if the interrupt condition exists. Also, if the process is at an equal or higher interrupt priority level, the interrupt is initiated when enabled.
-
- The *previous mode* field in the processor status longword is always set to *kernel* on an interrupt. On an exception, the field indicates the mode in which the exception occurred.
-

Processor Status

When an exception or interrupt is serviced, the processor status must be preserved. This is done so the interrupted process continues normally. Processor status preservation is the task of the program counter and the processor status longword. The counter and longword are restored with a *return from exception or interrupt* instruction. Any other status information needed to resume an interruptible instruction is stored in the general registers. Process context is not saved or restored on each exception or interrupt. Instead, context is saved and restored only when process context is switched. Other processors' status is changed less frequently.

There are several processor state variables associated with each process, and VAX software groups them into the 32-bit *processor status longword* (PSL). Bits 15 through 0 of the PSL are referred to separately as the *processor status word* (PSW). The PSW contains unprivileged information and those bits of the PSW that have defined meaning are controllable by any program. Bits 31 through 16 of the PSL have privileged status. While any program can perform the REI instruction (which loads PSL), the instruction refuses to load any PSL that would increase the privilege of a process, or create an undefined state in the processor. Figure 8-1 illustrates the processor status longword and the following paragraphs explain the various fields.

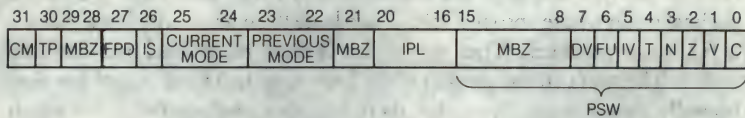


Figure 8-1 • Processor Status Longword

Bits 3:0 of the PSL are called the *condition codes*. In general, they reflect the result status of the most recent instruction that affects them. The condition codes are tested by the conditional branch instructions.

Bit 3 is the *negative* condition code (N bit). In general, it is set by negative result instructions. The bit is cleared by positive result or zero instructions. For those instructions that affect the bit according to a stored result, the N bit reflects the actual result even if the sign of the result is algebraically incorrect as a result of overflow.

Bit 2 is the *zero* condition code (Z bit). Typically it is set by instructions that store an exactly zero result and cleared if the result is not zero. Again, this reflects the actual result even if overflow occurs.

Bit 1 is the *overflow* condition code (V bit). In general, it is set after arithmetic operations in which the magnitude of the algebraically correct result is too large to be represented in the available space, and cleared after operations whose result fits. Instructions in which overflow is impossible or meaningless either clear the bit or leave it unaffected. Note that all overflow conditions that set the V bit can also cause traps if the appropriate trap enable bits are set.

Bit 0 is the carry condition code (C Bit). Usually, it is set after arithmetic operations in which a carry out of, or borrow into, the most significant bit occurred. The bit is cleared after arithmetic operations that had no carry or borrow, and is either cleared or unaffected by other instructions. The C bit is unique in that it not only determines the operation of conditional branch instructions, it also serves as an input variable to the ADWC (Add with Carry) and SBWC (Subtract with Carry) instructions used to implement multiple-precision arithmetic.

Bits 7 through 4 of the PSL are trap-enable flags that cause traps to occur under special circumstances.

Bit 7 is the *decimal overflow trap enable* bit (DV bit). When set, it causes a decimal overflow trap after the execution of any instruction that produces a decimal result whose absolute value is too large to be represented in the destination space provided. When the DV bit is clear, no decimal overflow trap occurs. The result stored consists of the low-order digits and sign of the algebraically correct result. Note that there are other trap conditions for which there are no enable flags—division by zero and floating overflow.

Bit 6 is the *floating underflow exception enable* bit (FU bit). When the FU bit is set (1), it forces a floating underflow exception after execution of the instruction that produced an underflowed result. When the FU bit is clear (0), no exception occurs. The result stored is zero.

NOTE

On VAX-11/780 processors with a hardware revision level of less than 7, a trap occurs. On all other VAX processors, a fault occurs.

Bit 5 is the *integer overflow trap enable* bit (IV bit). When set, it causes an integer overflow trap after an instruction that produced an integer result that could not be correctly represented in the space provided. When bit 5 is clear, no integer overflow trap occurs. The *V* condition code is set independently of the state of the *IV* condition code.

Bit 4 is the *trace* bit (T bit). When set, it causes a trace trap to occur after execution of the next instruction. This facility is used by debugging and performance analysis software to step through a program one instruction at a time. If any instruction is traced and causes an arithmetic trap, the trace trap occurs after the arithmetic trap.

Bits 15 through 8 of the PSL are not used and are reserved.

Bits 20 through 16 represent the processor's *interrupt priority level* (IPL). In order to be acknowledged by the processor, an interrupt must be at a priority higher than the current IPL. Virtually all software runs at IPL 0, so the processor acknowledges and services interrupt requests of any priority. The interrupt service routine for any request runs at the IPL of the request. This temporarily blocking interrupt requests lower or equal priority. Briefly, there are 31 interrupt priority levels above zero, numbered 01 through 1F (hexadecimal). Interrupt levels 01 through 0F exist entirely for use by software. IPLs 10 through 17 are for use by peripheral devices and their controllers. Although present systems support only 14 through 17. Levels 18 to 1F are for use for urgent conditions including the interval clock, serious errors, and *powerfail*.

Bits 23 and 22 are the *previous mode* bits that contain the value from the current mode field at the most recent exception that transferred from a less privileged mode to this one. Previous mode is of interest in the PROBE instructions that enable privileged routines to determine whether a caller at the previous mode is sufficiently privileged to reference a given area of memory.

Bits 25 and 24 are the *current mode* bits that determine the privilege level of the currently executing program. Privilege is granted in two ways by the mode field—certain instructions (*halt*, *move to processor register*, and *move from processor register*) are not performed unless the current mode is kernel. The memory management logic controls access to virtual addresses on the basis of the program's current mode, the type of reference (read or write), and a protection code assigned to each page of the address space.

Bit 26 is the *interrupt stack* flag (IS bit) that indicates that the processor is using the special *interrupt stack* rather than one of the four stacks associated with the current process. When the IS bit is set, the current mode is always *kernel*. Thus, for example software operating on the *interrupt stack* has full kernel mode privileges.

Bit 27 is the *first part done* flag (FPD bit) that the processor uses in certain instructions. These instructions may be interrupted or page faulted in the middle of their execution. If the FPD bit is set when the processor returns from an exception or interrupt, the processor resumes the interrupted operation where it left off rather than restart the instruction.

Bit 30 is the *trace pending* bit (TP bit) that is used by the processor to ensure that one trace trap occurs for each instruction performed with the *trace* bit set.

Bit 31 is the *compatibility mode* bit (CM bit). When the CM bit is set, the processor is in PDP-11 compatibility mode and executes PDP-11 instructions. When the bit is clear, the processor is in native mode and executes VAX instructions. Compatibility mode may be omitted from subset implementations of the VAX architecture. In a processor that does not have compatibility mode, this bit is always clear.

Asynchronous System Traps

An asynchronous system trap (AST) is used to notify a process that some events are not synchronized with process execution. Traps are also used to initiate processing for those events with the least possible delay.

Delay in delivery may be due to one of two causes. Either the process is not on the system or there is an access mode mismatch. The efficient handling of traps in VAX processors requires some hardware assistance to detect changes in access mode.

Each of the execution access modes may receive ASTs. However, an AST for a less privileged access mode must not be permitted to interrupt execution in a more privileged access mode. Because transitions to a less privileged access mode occur only in the *return from exception or interrupt* instruction, comparison of the current access mode field is made with a privileged register (ASTLVL). The register contains the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an Interrupt Processor Level (IPL) 2 interrupt is initiated to deliver the pending AST.

NOTE

Loading an ASTLVL or LDPCTX instruction with a *move to processor register* instruction does not request a software interrupt at IPL2. During a *return from exception or interrupt* instruction only can an ASTLVL instruction cause an interrupt request.

The general software flow for AST processing is described in the following paragraphs.

1. An event associated with an AST causes software to put an AST control block in the queue to the software process control block. Then the software sets the hardware process control block ASTLVL field to the most privileged access mode for which an AST is pending. If the target process is executing, the ASTLVL privileged register also has to be set.
2. When a *return from exception or interrupt* instruction detects a transition to an access mode that can be interrupted by a pending AST, a priority level 2 interrupt is requested to deliver the AST. Note that the instruction does not check pending ASTs when returning to a routine executing on the interrupt stack.
3. The priority level 2 interrupt service routine computes the new value for ASTLVL to prevent additional AST delivery interrupts while in kernel mode. And the service routine moves that value to the process control block and the ASTLVL register before lowering the interrupt priority level and actually dispatching the AST. This interrupt service routine normally executes on the kernel stack in the context of the process receiving the AST.
4. At the conclusion of processing for an AST, the ASTLVL is recomputed and moved to the process control block and ASTLVL register by software.

NOTE

Two of the software interrupt priority levels are reserved for process structure software. Interrupt priority level 2 is for AST delivery interrupts. Interrupt priority level 3 is for process scheduling interrupts.

▪ Exceptions

There are six types of exceptions—arithmetic, instruction fault, memory management, operand reference, serious system failures, and tracing. All are described in the subsequent paragraphs.

Arithmetic Exceptions

Exceptions caused by arithmetic or conversion operations are mutually exclusive and can be assigned the same vector in the system control block. Each indicates that an exception occurred during the last instruction and that the instruction has been either completed (in the case of a trap) or *backed up* (in the case of a fault). A code identifying the exception is written on the stack as a longword. Figure 8-2 illustrates the stack after an arithmetic exception. In the case of a fault, the program counter of the next instruction is the same as the instruction that caused the exception. Arithmetic exception codes are listed in Table 8-1.

TYPE CODE
PROGRAM COUNTER OF NEXT INSTRUCTION TO EXECUTE
PROCESSOR STATUS LONGWORD

Figure 8-2 ▪ *Stack after Arithmetic Exception*

Table 8-1 • Arithmetic Exception Type Codes

Code	Exception Type	Software Mnemonic
Traps		
1	Integer overflow	SRM\$K__ - INT__ OVF__ T
2	Integer divide by zero	SRM\$K__ INT__ DIV__ T
3	Floating overflow*	SRM\$K__ FLT__ OVF__ T
4	Floating/decimal divide by zero	SRM\$K__ FLT__ DIV__ T
5	Floating underflow*	SRM\$K__ FLT__ UND__ T
6	Decimal overflow	SRM\$K__ DEC__ OVF__ T
7	Subscript range	SRM\$K__ SUB__ RNG__ T
Faults		
8	Floating overflow	SRM\$K__ FLT__ OVF__ F
9	Floating divide by zero	SRM\$K__ FLT__ DIV__ F
10	Floating underflow	SRM\$K__ FLT__ UND__ F

* Not on VAX-11/750

An *integer overflow trap* exception indicates that the preceding instruction set the overflow condition code bit. This trap occurs only if the integer overflow enable bit in the processor status word is set. The result stored is the low-order part of the correct result, and type code 1 is put on the stack. The negative and zero condition-code bits are set according to the stored result. Note that the BISPSW, MOVTUC, REI, REMQHI, REMQTI, REMQUE, and RET instructions do not cause an integer overflow even if they set the overflow condition code bit. EMOD instructions can cause integer overflow.

An *integer divide by zero trap* exception indicates that the preceding instruction had an integer zero divisor. The result stored is equal to the dividend, and the overflow condition code bit is set and type code 2 is put on the stack.

A *decimal string divide by zero trap* exception indicates that the preceding instruction had a decimal string zero divisor. The destination, registers R0 through R5, and condition codes are *unpredictable*. The zero divisor can be either positive or negative. Type code 4 is put on the stack.

A *decimal string overflow trap* exception indicates that the preceding instruction had a decimal string result too large for the destination string provided, and that decimal overflow trap enable bit is set. The overflow condition code bit is always set. Type code 6 is put on the stack.

A *subscript range trap* exception indicates that the preceding instruction was an index instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in the *indexout* operand and the condition codes are set as if the operand is within range. Type code 7 is put on the stack.

A *floating-overflow fault* exception indicates that the preceding instruction resulted in an exponent greater than the largest representable exponent for the data type. The result is normalized and rounded before comparison. The destination is unaffected and the saved condition codes are *unpredictable*. The saved program counter points to the instruction causing the fault. If the instruction is an extended polynomial instruction, it is suspended and the processor status word *first part done* bit is set. Type code 8 is put on the stack.

A *divide by zero floating fault* exception indicates the preceding instruction had a floating zero divisor. The quotient operand is unaffected, and the saved condition codes are *unpredictable*. The saved program counter points to the instruction causing the fault. Type code 9 is put on the stack.

A *floating-underflow fault* exception indicates that the preceding instruction resulted in an exponent less than the smallest representable exponent for the data type. The result is normalized and rounded before comparison. The destination operand is unaffected, and the saved condition codes are *unpredictable*. The saved program counter points to the instruction causing the fault. If the instruction is an extended polynomial instruction, it is suspended and the processor status word's *first part done* bit is set. Type code A is put on the stack.

Instruction Fault

There are four instruction faults. They are *breakpoint* fault, *compatibility mode* fault, *opcode reserved to Digital* fault, and *opcode reserved to users* fault.

A *breakpoint* fault occurs when the breakpoint (BPT) instruction is executed. No parameters are saved. To proceed from a breakpoint, a debugger or tracing program typically restores the original contents of the location containing the breakpoint, sets the trace enable bit in the processor status longword that was saved by the breakpoint fault, and resumes. When the breakpointed instruction completes, a trace exception occurs. Then the tracing program can reinsert the breakpoint, restore the trace enable bit of the processor status longword to its original state, and resume. Note that if both tracing and breakpointing are in progress, then on the trace exception both the breakpoint restoration and a normal trace exception should be processed by the trace handler.

A *compatibility mode* fault occurs when the processor is in compatibility mode. A longword of information is written to the stack. All other exceptions in compatibility mode occur to the regular VAX vector. The compatibility mode is an option and is not present on all VAX systems.

An *opcode reserved to Digital* fault occurs when the processor finds an opcode that is not specifically defined, or one that requires higher privileges than the current mode. No parameters are written. Opcode FFFF (hexadecimal) is always faulted.

An *opcode reserved to users* fault occurs for exactly the same reasons as above except that the event is caused by a different set of opcodes and faults through a different vector. All user-reserved opcodes start with FC (hexadecimal). If the special instruction needs to generate a unique exception, one of the user-reserved vectors should be used. An example of a unique exception is an unrecognized second byte of an instruction.

Memory Management Exceptions

There are two memory management exceptions—the *access control violation* fault, and the *translation not valid* fault.

An *access control violation* fault is an exception indicating that the process attempted a reference not allowed at the access mode at which the process was operating. Software may restart the process after changing the address translation information.

A *translation not valid* fault indicates the process attempted a reference to a page for which the *valid* bit of the page table was not set. If a process attempts to reference a page for which the page table entry specifies both *not valid* and *access violation*, an *access control violation* fault occurs.

Operand Reference Exceptions

Two types of operand reference cause exceptions—a *reserved addressing mode* fault, and a *reserved operand* exception.

A reserved addressing mode fault is an exception that indicates an operand specifier attempted to use an addressing mode that is not allowed. No parameters are written. A *short literal* specifier is not allowed in the modify, destination, address source operand reference. A *register* specifier is not allowed in an address source operand reference. An index mode specifier is not allowed with the program counter as the index. Short literal, register and index mode specifiers are not allowed in the index mode.

A *reserved operand* exception indicates that an accessed operand has a format reserved for future use by Digital. No parameters are written. This exception always *backs up* the program counter to point to the opcode. The exception service routing may determine the type of operand by examining the opcode using the stored program counter. Note that only the changes made by instruction fetch and, because of operand specifier evaluation, may be restored. Therefore, some instructions are not restartable. These exceptions are labeled as *aborts* rather than *faults*. The program counter is always restored properly unless the instruction attempted to modify the counter so that it has unpredictable results. With the exception of the *first part done* and the *trace pending* bits, the processor status longword is not changed except for the condition codes, which are *unpredictable*. Reserved operand exceptions are caused by the following conditions.

-
- Bit field is too wide.
-
- Decimal string is too long.
-
- Floating-point numbers with the sign bit set and the exponent is zero except in the POLY table.
-
- Floating-point numbers with the sign bit set and the exponent is zero in the POLY table.
-
- Incorrect source string length at completion of EDITPC instruction.
-
- Invalid bit combination in a BISPSW or BICPSW instruction.
-
- Invalid bit combination in PSW or MASK longword during RET instruction.
-
- Invalid CALL entry mask.
-
- Invalid combinations in the process control block loaded by an LDPCTX instruction.
-
- Invalid digit in a CVTTP or CVTSP instruction.
-
- Invalid operand addresses in an INSQHI, INSQTI, REMQHI, or REMQTI instruction.
-
- Invalid processor status longword bit combination stored by a *return from exception or interrupt* instruction.
-
- Invalid register content in MTPR instructions to some register for some implementations.
-
- Invalid register number in MFPR or MTPR instruction.
-

-
- Misaligned operand in ADAWI instruction.
-
- POLY degree is too large.
-
- Reserved pattern operator in EDITPC instruction.
-

Serious System Failures

Serious system failures are processed by privileged software. There are three types of serious system failures.

-
- Interrupt stack not valid—halt.
-
- Kernel stack not valid—abort.
-
- Machine check.
-

An *interrupt stack not valid—halt* indicates that

-
- The interrupt stack was invalid.
-
- A memory error occurred while the processor was writing information onto the stack during the initiation of an exception or interrupt.
-

No further interrupt requests are acknowledged on this processor. The processor leaves the program counter, the processor status longword, and the reason for the halt in registers. That is made available to a debugger, the normal bootstrap routine, or an optional watchdog bootstrap routine. A watchdog bootstrap routine can cause the processor to leave the halted state.

Kernel stack not valid—abort exceptions indicate that the kernel stack was not valid while the processor was writing information onto the stack during the initiation of an exception or interrupt. Usually, this is an indication of stack overflow or another executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No information other than the processor status longword and program counter is written onto the interrupt stack. The interrupt priority level is raised to 1F (hexadecimal). Software may abort the process without aborting the system. Because of the lost information, the process cannot be continued. If the kernel stack is not valid during the normal execution of an instruction including *change mode kernel* and *return from exception or interrupt* instructions, the processor initiates the normal memory management fault. If the exception vector for kernel stack not valid is 0 or 3, the behavior of the processor is *undefined*.

A *machine check* exception indicates that the processor detected an internal error. Machine check exceptions can be caused by such bus errors as nonexistent memory, cache parity, translation buffer parity, or by a control store parity error. Like other exceptions, this exception is taken independently of the interrupt priority level. The level is raised to 1F (hexadecimal). Implementation-specific data is written as longwords to the stack. The processor specifies the length parameter by placing the number of bytes written as the last longword written. This count excludes the program counter, processor status longword, and the length parameter. On the basis of presented information, software decides whether or not to abort the current process if the machine check came from the process. Machine check includes uncorrected bus and memory errors, and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state is preserved on a *best effort* basis. If the exception vector for the *machine check* is 0 or 3, the behavior of the processor is *undefined*. Under these conditions, the VAX processor halts.

Trace Exceptions

Trace exceptions occur between instructions when trace is enabled. Trace is used for tracing programs, for performance evaluation, or for debugging purposes. The machine is designed so that one trace exception occurs before the execution of each traced instruction. The program counter saved on a trace is the address of the next instruction that would normally be executed. If a trace fault and a memory management fault occur simultaneously, the order in which the exceptions are taken is *unpredictable*. The trace fault for an instruction takes precedence over all other exceptions.

In order to ensure that exactly one trace occurs per instruction despite other traps and faults, the processor status longword contains two bits—the trace enable (T) bit, and the trace pending (TP) bit. If only one bit is used, the occurrence of an interrupt at the end of an instruction would produce either no trace or two traces depending on the design. The trap is implemented by copying the trace enable bit to a second bit. The second bit is the trace pending (TP) bit. The TP bit is used to generate the exception. The trace pending bit generates a fault before any other processing at the start of the next instruction.

The rules of operation for trace are as follows:

1. At the beginning of an instruction, if the trace pending bit is set, a trace fault is taken after clearing the trace pending bit.
2. The trace pending bit is loaded with the value of the trace bit.
3. If the instruction faults or an interrupt is serviced, the trace pending bit is cleared before writing the processor status longword. The written program counter is set to the start of the faulting or interrupted instruction. Instruction execution is resumed at step 1 above.
4. If the instruction aborts or takes an arithmetic trap, the trace pending bit of the processor status longword is not changed before the processor status longword is written.
5. If an interrupt is serviced after instruction completion and arithmetic traps but before tracing is checked for at the start of the next instruction, then the trace pending bit is not changed before the processor status longword is written.

The routine entered by a change mode instruction is not traced because the instruction clears the trace and trace pending bits in the new processor status longword. However, if the trace bit was set at the beginning of the change mode instruction, the trace and trace pending bits of the saved processor status longword are set. Trace faults resume with the instructions following the *return from exception or interrupt* (REI) instruction in the routine that was entered by the change mode instruction.

An instruction following a REI faults either if the trace bit is set when the REI instruction was executed, or if the trace pending bit is set in the saved processor status longword. In both cases, the trace pending is set after the REI instruction. Note that a trace fault is taken with the new processor status longword if that fault occurs for an instruction following a return from exception or interrupt that sets the trace pending bit. Thus, special care must be observed if exception or interrupt routines are traced. If the trace bit is set by a BISPSW instruction, trace faults begin with the second instruction after the BISPSW instruction.

In addition, the *call* instructions save a *clear trace* bit, although the *trace* bit in the processor status longword is unchanged. This is done so that a debugger or trace program proceeding from a breakpoint fault does not get a spurious trace from the RET instruction that matches the *call* instruction.

The detection of reserved instruction faults occurs after the trace fault. The detection of interrupts and other exceptions can occur during instruction execution. In this case, the trace pending bit is cleared before the exception or interrupt is initiated. The entire processor status longword is saved automatically on interrupt or exception initiation and is restored at the end with an REI instruction. This makes interrupts and benign exceptions totally transparent to the executing program.

Routines using the trace facility are called trace handlers. When developing handling routines, the following conventions and restrictions should be observed.

1. When the trace handler routine returns control to the traced program, the handler should always set the trace bit of the processor status longword that is to be restored. This prevents other programs from clearing the bit.
2. The trace handler should never examine or alter the trace pending bit when tracing. The hardware ensures that this bit is correctly maintained.
3. When tracing is complete, both the trace and trace pending bits must be cleared. This ensures that tracing ceases.
4. Tracing a service routine that completes with an REI instruction initiates a trace in the restored mode when the instruction completes. If the program to which control is being restored was being traced, only one trace exception is initiated.
5. If a routine entered by a *call* instruction is executed at full speed by clearing the trace bit, trace control can be regained by setting the trace bit in the call frame of the processor status word. Tracing resumes after the instruction following the RET instruction.
6. Tracing is disabled for routines entered by a *change mode* instruction or any exception. If a *change mode* instruction or exception service routine is to be traced, a breakpoint instruction must be placed at the entry point in the routine. If the routine is recursive, breakpointing catches each recursion only if the breakpoint is not on the *change mode* instruction or the instruction with the exception.
7. If multiple trace handlers are used, all handlers must preserve the trace bit when turning the handler on and off. They also have to simulate traced code that alters or reads the *trace* bit.

▪ Interrupts

The processor arbitrates interrupt requests according to priority. When the interrupt request priority level is higher than the current interrupt priority level, the processor raises the interrupt priority level and services the interrupt request. The interrupt service routine is entered at the interrupt priority level of the interrupt request and usually does not change the interrupt priority level set by the processor.

Interrupt requests can come from devices, controllers, other processors, or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing an MTPR instruction with the source operand specifying the priority desired. However, a processor cannot disable interrupts on other processors. Furthermore, the priority level of one processor does not affect the priority level of the other processors. Thus, in multiprocessor systems, interrupt priority levels cannot be used to synchronize access to shared resources. Even the various urgent interrupts including those exceptions that run at IPL 1F (hexadecimal) do so on one processor only. Because of this, special software action is required to stop other processors in a multiprocessor system.

The processor services an interrupt request when the currently executing instruction is completed. The processor also services interrupt requests at well-defined points during the execution of long, iterative instructions. To avoid saving additional instruction state in memory, interrupts are initiated when the instruction state can be completely contained in the registers, processor status longword, and program counter. The following events cause interrupts:

-
- Asynchronous system trap delivery when a *return from exception or interrupt* instruction restores a processor status longword with the interrupt stack bit clear, and mode greater than or equal to ASTLVL (IPL 2 (hexadecimal)).
-
- Console storage device (IPL 17 (hexadecimal) or IPL 14 (hexadecimal)).
-
- Console terminal transmit and receive (IPL 14 (hexadecimal)).
-
- Device alert (IPL 10:17 (hexadecimal)).
-
- Device completion (IPL 10:17 (hexadecimal)).
-
- Device error (IPL 10:17 (hexadecimal)).
-
- Device memory error (IPL 10:17 (hexadecimal)).
-
- Interval timer (IPL 18 (hexadecimal)).
-

-
- Power failure (IPL 1E (hexadecimal)).
-
- Recovered memory, bus or processor errors (the VAX-11/750 interrupts at IPL 1A (hexadecimal) for corrected memory reads; the VAX-11/780 at IPL 1B (hexadecimal); implementation specific).
-
- Software interrupt invoked by a *move to processor register* instruction with the software interrupt request register as the destination (IPL 1F (hexadecimal)).
-
- Unrecovered memory, bus, or processor errors (the VAX-11/750 and VAX-11/780 interrupt at IPL 1D (hexadecimal) for write memory errors, implementation specific).
-

Each device controller has a separate set of interrupt vector locations in the system control block. This eliminates the need to determine which controller originated the interrupt. The vector address for each controller is fixed by hardware.

In order to reduce interrupt overhead, memory mapping information is not changed when an interrupt occurs. The instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.

Device Interrupts

Interrupt priority levels 10 through 17 (hexadecimal) are assigned to device interrupts. Any given implementation may or may not have all levels of interrupts. For example, on the VAX-11/750, levels 14 (hexadecimal) through 17 (hexadecimal) only are available for device interrupts. These levels correspond to the UNIBUS levels BR4 through BR7.

Software-generated Interrupts

The system software has 15 interrupt priority levels (1 through F (hexadecimal)). Refer to the *VAX Software Handbook* for details of these interrupts. Two registers are used in software-generated interrupt processing—the software interrupt summary register, and the software interrupt request register.

The software interrupt summary register (SISR) is a privileged register that records pending software interrupts. The register contains a value of 1 in the bit positions corresponding to levels on which software interrupts are pending. See Figure 8-3. All such levels must be lower than the current processor interrupt priority level. Otherwise, the processor would have taken the requested interrupt.

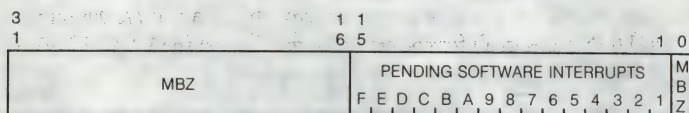


Figure 8-3 ■ Software Interrupt Summary Register

The software interrupt summary register is a read/write register accessible only to privileged software. During bootstrap procedures, the contents of the register are cleared. To read the contents of the register, use the *move from processor register* instruction. To write to the register, use the *move to processor register* instruction. The *move to processor register* instruction writes to the register; but this is not the normal way to make software interrupt requests. The instruction is useful for clearing the software interrupt system and for reloading its state after a power failure.

The software interrupt request register is a write-only 4-bit privileged register used for making software interrupt requests. See Figure 8-4. Executing an *MTPR source, #SIRR* instruction requests an interrupt at the level specified by bits 0 through 3 of the source operand. After a software interrupt request is made, the corresponding bit in the register is set. The hardware clears the bit when the interrupt is taken. If the specified level is greater than the current interrupt priority level, the interrupt occurs before execution of the following instruction. If the specified level is less than or equal to the current interrupt priority level, the interrupt is deferred until the interrupt priority level is lowered to less than the specified level with no higher interrupt level pending. Either a *return to exception or interrupt* or a *move to processor register* instruction lowers the level. If the value of bits 0 through 3 of the specified source is 0, an interrupt does not occur.

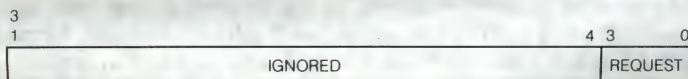


Figure 8-4 ■ Software Interrupt Request Register

No indication is given if there is a request at the selected level. Therefore, the service routine must not assume there is a one-to-one correspondence of interrupts generated to interrupts initiated. A valid protocol for generating such a correspondence is as follows:

-
- The requester uses an INSQUE instruction to replace a control block describing the request onto a queue for the service routine.
-
- The requester uses an MTPR instruction to request an interrupt at the appropriate level.
-
- The service routine uses a REMQUE instruction to remove a control block from the queue of the service requests. If the instruction returns *failure* (nothing in the queue), the service routine exits with a *return from exception or interrupt* instruction.
-
- If the REMQUE instruction returns with an item from the queue, the service routine performs the service and returns to the third step to look for other requests.
-

Urgent Interrupts

The processor has eight priority levels for urgent conditions including serious errors and *powerfail*. Interrupts on these levels are initiated by the processor upon detection. Some of these conditions are not interrupts. For example, a *machine check* is usually an exception. But it runs at a high priority level on the interrupt stack. Interrupt level 1E (hexadecimal) is reserved for *powerfail*. Interrupt level 1F (hexadecimal) is reserved for those exceptions that must lock out all processing until handled. This includes the hardware and software *disasters* (*kernel stack not valid* and *machine check*). It might also be used to allow a kernel mode debugger to gain control on any exception.

Interrupt Priority Level Register

Writing to the interrupt priority level register with the *move to processor register* instruction loads the processor priority field in the processor status longword. That is, bits 20 through 16 of the processor status longword are loaded from bits 4 through 0 of the interrupt priority level register. Reading from the interrupt priority level register with the *move from processor register* instruction reads the processor priority field from the processor status longword. When writing to the register, bits 5 through 31 are ignored. When reading from the register, bits 5 through 31 are returned zero. During a bootstrap routine, the interrupt priority level is initialized to 1F (hexadecimal).

Interrupt service routines must follow the discipline of not lowering the interrupt priority level below the initial level. If they do, an interrupt at an intermediate level could cause improper stack nesting. This would fault the *return from exception or interrupt* instruction. Actually, a service routine could lower the interrupt priority level if it ensured that no intermediate levels could interrupt. However, this would result in *unreliable* code.

Interrupt Example

Using Example 8-1, assume the processor is running in response to an interrupt at interrupt priority level 5 (Step 1). (All numbers in this example are hexadecimal.) Then the processor sets the interrupt priority level to 8 (Step 2) and posts software requests at interrupt priority levels 3 (Step 3), 7 (Step 4), and 9 (Step 5). Subsequently, a device interrupt arrives at interrupt priority level 11 (Step 6). Finally the interrupt priority level is set back to interrupt priority level 5 (Step 10).

Example 8-1 ■ Interrupt Sequence

Step	Event	State after Event Interrupt Priority Level in Contents of SISR PSL		
		IPL (hex)	(hex)	stack
1	Initiate sequence	5	0	0
2	MTPR #8, #IPL instruction	8	0	0
3	MTPR #3, #SIRR instruction	8	8	0
4	MTPR #7, #SIRR instruction	8	88	0
5	MTPR #9, #SIRR instruction	9	88	8,0
6	Interrupts to device	11	88	9,8,0
7	Interrupts to device service routine REI	9	88	,8,0
8	IPL 9 service routine REI	8	88	0
9	MTPR #5, #IPL instruction changes IPL to 5 and the request for 7 is granted immediately	7	8	5,0
10	IPL 7 service routine REI	5	8	0
11	Initial IPL 5 service routine REI back to IPL 0 and the request for 3 is granted immediately	3	0	0
12	IPL 3 service routine REI	0	0	-

▪ System Control Block

The system control block (SCB) is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines. The system control block base is a privileged register containing the physical address of the system control block, which must be page-aligned. During bootstrap routines, the contents of the system control block base register are *unpredictable*. The actual length is dependent upon the system implementation because the length represents a physical address.

NOTE

On some processors, the SCB may have additional pages that contain the addresses of interrupt service routines for I/O devices.

A vector is a longword in the SCB that is examined by the processor when an exception or interrupt occurs. The vector is used to determine how to service the event. See Table 8-2 for a list of the vectors. Separate vectors are defined for each interrupting device controller and each class of exception. Each vector is interpreted according to the value stored in bits 0 and 1.

If the value is 0, the event is serviced on the kernel stack unless it is running on the interrupt stack. If it is running on the interrupt stack, it is serviced there. Behavior of the processor is *undefined* for a *kernel stack not valid* exception with this code.

If the value is 1, the event is serviced on the interrupt stack. If this event is an exception, the interrupt priority level is raised to 1F (hexadecimal).

If the value is 2, the event is serviced in writable control store passing bits 2 through 15 to the installation-specific microcode there. If writable control store does not exist or is not loaded, the operation is *undefined*.

If the value is 3, the operation is *undefined*.

For values 0 and 1, bits 2 through 31 contain the virtual address of the service routine. The address must begin on a longword boundary and is normally in the system space. A change mode instruction is serviced on the stack selected by the new mode. Bits 0 and 1 in the change mode vectors must be zero or the operation is *undefined*.

Table 8-2 ■ Event Vectors

Vector (hex)*	Vector Name	Type of Event	Number of Parameters	Notes
00	Passive Release	Interrupt		May occur when an interrupt request is removed before the interrupt is initiated. IPL is that of the request.
04	Machine Check	Abort or Fault or Trap	†	Processor- and error-dependent information is pushed onto the stack if possible. Restartability is processor-dependent. IPL is raised to 1F(hex) and the interrupt stack is used ($PSL < IS > \leftarrow 1$).
08	Kernel Stack Not Valid	Abort	0	IPL is raised to 1F(hex) and the interrupt stack is used ($PSL < IS > \leftarrow 1$).
0C	Powerfail	Interrupt	0	IPL is raised to 1E(hex)
10	Reserved or Privileged Instruction	Fault	0	Opcodes reserved to Digital and privileged instructions
14	Customer-reserved Instruction	Fault	0	XFC instruction
18	Reserved Operand	Fault or Abort	0	Type depends on the circumstances
1C	Reserved Addressing Mode	Fault	0	
20	Access Control Violation	Fault	2	Virtual address causing fault is pushed onto stack

Notes:

* (Hex) indicates the preceding number is in hexadecimal notation.

† The number of bytes of parameters is pushed onto the stack and is implementation-dependent.

Table 8-2 • Event Vectors (Cont.)

Vector (hex)*	Vector Name	Type of Event	Number of Parameters	Notes
24	Translation Not Valid	Fault	2	Virtual address causing the fault is pushed onto the stack
28	Trace Pending	Fault	0	
2C	Breakpoint Instruction	Fault	0	
30	Compatibility	Fault or Abort	1	A type code is pushed onto the stack
34	Arithmetic	Trap or Fault	1	A type code is pushed onto the stack
38:3C	Reserved to Digital			
40	CHMK	Trap	1	The operand word is sign extended and pushed onto the stack. Vector <1:0> must be zeros.
44	CHME	Trap	1	The operand word is sign extended and pushed onto the stack. Vector <1:0> must be zeros.
48	CHMS	Trap	1	The operand word is sign extended and pushed onto the stack. Vector <1:0> must be zeros.
4C	CHMU	Trap	1	The operand word is sign extended and pushed onto the stack. Vector <1:0> must be zeros.

Table 8-2 • Event Vectors (Cont.)

Vector (hex)*	Vector Name	Type of Event	Number of Parameters	Notes
50:60	Reserved for System Bus and Memory Errors	Interrupt		IPL is implementation dependent.
64:80	Reserved to Digital			
84	Software Level 1	Interrupt	0	IPL is 1.
88	Software Level 2	Interrupt	0	IPL is 2. Ordinarily used for AST delivery.
8C	Software Level 3	Interrupt	0	IPL is 3. Ordinarily used for process scheduling.
90:BC	Software Levels 4:F	Interrupt	0	Vector corresponds to IPL.
C0	Interval Timer	Interrupt	0	IPL is 16 or 18(hex).
C4	Reserved to Digital			
C8	Subset Emulation	Trap	10	FPD bit clear. Subset VAX systems only.
CC	Suspended Emula- tion	Fault	0	FPD bit set. Subset VAX systems only.
D0:DC	Reserved to Digital			
E0:EC	Reserved to Cus- tomer or Computer Special Systems (Digital)			
F0	Console Storage Receive	Interrupt	0	On VAX-11/730 and VAX- 11/750. IPL is implementa- tion-dependent.

Table 8-2 • Event Vectors (Cont.)

Vector (hex)*	Vector Name	Type of Event	Number of Parameters	Notes
F4	Console Storage Transmit	Interrupt	0	On VAX-11/730 and VAX-11/750 only. IPL is implementation-dependent.
F8	Console Terminal Receive	Interrupt	0	IPL is 14(hex).
FC	Console Terminal Transmit	Interrupt	0	IPL is 14(hex).
100:13C	Adapter Vectors	Interrupt	0	IPL is 14(hex).
140:17C	Adapter Vectors	Interrupt	0	IPL is 15(hex).
180:1BC	Adapter Vectors	Interrupt	0	IPL is 16(hex).
1C0:1FC	Adapter Vectors	Interrupt	0	IPL is 17(hex).
200:3FC	Device Vectors	Interrupt	0	May be any IPL 14:17(hex).
400:5FC	Device Vectors	Interrupt	0	May be any IPL 14:17(hex).

▪ Stacks

The processor is either in a process context or a systemwide interrupt service context at all times. When in the process context, the processor is in one of four modes (kernel, executive, supervisor, or user), and the interrupt stack (IS) is zero. When the processor is in the systemwide interrupt service context, it operates with kernel privileges, and the interrupt stack is one. A stack pointer (SP) is assigned to each of these five states. Whenever the processor changes states, stack pointer R14 is stored in the process context stack pointer for the old state and loaded from that for the new state. The process context stack pointers are allocated in the hardware process control block. There are four stack pointers—KSP (kernel), ESP (executive), SSP (supervisor), and USP (user).

Operating system design must choose a priority level that is the boundary between kernel and interrupt stack use. The system control block interrupt vectors must be set so the interrupts to levels above the boundary run on the interrupt stack and interrupts below this boundary run on the kernel stack. Typically, asynchronous system trap delivery is on the kernel stack and higher levels are on the interrupt stack.

In addition, VAX systems keep copies of the four process stack pointers in privileged registers. These registers are accessed during stack switch operations. The stack pointers in the hardware process control block are referenced only during context switch by the *save process context* (SVPCTX) and *load process context* (LDPCTX) instructions.

Stack Location

The executive, supervisor, and user stacks need not be resident in main memory. The kernel can bring in or allocate process stack pages as *address translation not valid* faults occur. However, the kernel stack for the current process and the interrupt stack must be resident and accessible. *Translation not valid* and *access control violation* faults occurring on references to either of these stacks are serious system failures from which recovery is impossible.

If either of these faults occurs on a kernel stack reference, the processor aborts the current sequence and initiates a *kernel stack not valid* abort on hardware level 1F (hexadecimal). If either fault occurs on a reference to the interrupt stack, the processor halts. Note that this does not mean every possible reference is checked. It means the processor does not loop under these conditions. The kernel stack for processes other than the current one need not be resident; but it must be resident before the software's process dispatcher selects a process to run. Further, any mechanism using *access control violation* or *translation not valid* faults to gather process statistics must exercise care not to invalidate kernel stack pages.

Stack Alignment

Except on *call* instructions, the hardware does not attempt to align the stacks. For best performance, the software should align the stack on a longword boundary and allocate the stack in longword increments. In order to keep the stacks longword-aligned, the following six instructions are recommended.

-
- Convert byte to longword (CVTBL).
 - Convert longword to byte (CVTLB).
 - Convert longword to word (CVTLW).
 - Convert word to longword (CVTWL).
-

-
- Move zero-extended byte to longword (MOVZBL).
 - Move zero-extended word to longword (MOVZWL).
-

Status Bits

The interrupt stack bit and current mode bits in the processor status longword (PSL) specify which of the five stack pointers is in use. Table 8-3 lists the interrupt stack (IS) and current mode bits that identify the stack pointers.

Table 8-3 • PSL Stack Status Bits

IS Bit	Current Mode Bit	Register
1	0	Interrupt stack pointer (ISP)
0	0	Kernel stack pointer (KSP)
0	1	Executive stack pointer (ESP)
0	2	Supervisor stack pointer (SSP)
0	3	User stack pointer (USP)

The processor does not allow the current mode bits to be set (1) when the interrupt stack bit is set. This is done by clearing the mode bits

-
- When taking an exception or interrupt.
 - By causing a reserved operand fault if the *return from exception or interrupt* instruction attempts to load a processor status longword in which both the interrupt status bit and current mode bits are set.
-

The stack to be used for an exception is selected by the current processor status longword interrupt stack bit and the event vector bits. Figure 8-5 illustrates the stack selection logic. Values 10 (binary) and 11 (binary) of the vector are used for other purposes as described in the system control block vectors section.

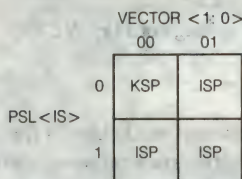


Figure 8-5 • Stack Selection

Accessing Stack Registers

Reference to the stack pointer in the general registers will access one of five stack pointers depending on the values of the *current mode* and *interrupt stack* bits in the processor status longword. Some processors implement stack pointers as processor registers. On these processors, software can access any of the stack pointers that are not selected by the *current mode* and *interrupt stack* bits. Results are correct even if the currently selected stack pointer is referenced in the processor register space by an MTPR or MFPR instruction. If the process stack pointers are implemented as registers, move processor register instructions are the only method for accessing the stack pointers of the current process. If the process stack pointers are kept only in the process control block, an MTPR or MFPR instruction might not access the process control block.

The internal processor register numbers were chosen to be the same as bits 24 through 26 of the processor status longword. The previous stack pointer is the same as bits 22 and 23 of the processor status longword unless the *interrupt stack* bit is set. If the *interrupt stack* bit is set, the previous mode cannot be determined from the processor status longword because interrupts always clear bits 22 and 23 of that longword. At bootstrap time, the contents of all stack pointers are *unpredictable*.

▪ Recognition Priority

The order in which recognition of simultaneous exceptions and interrupts takes place is as follows:

1. Arithmetic exceptions.
2. Console halt or higher priority interrupt.
3. Machine check exception.

4. Start instruction execution or restart suspended instruction.
5. Trace fault (only one per instruction).

NOTE

The order in which console halt and interrupt recognition occurs is not dictated by the VAX architecture. Future VAX processors may not take these in the same order as the VAX-11/750 or VAX-11/780 that take console halts before interrupts.

▪ Suspended Instructions

The VAX architecture allows the suspension of certain instructions at well-defined intermediate points in the execution. This is done to take memory management faults, console halts, or interrupts. In this case, the hardware uses processor status longword trace and trace pending bits to ensure that no additional trace faults occur when execution is resumed.

▪ Initiating an Exception or Interrupt

The handling of the event is determined by the contents of the longword vector in the system control block. If bits 0 and 1 of the vector contain an *invalid* code, the CPU behavior is *unpredictable*.

During the following sequence, interrupts are disabled.

1. The condition codes are replaced with zeros if bits 0 and 1 of the vector have a value of 0 or 1.
2. The current pointer is saved and the new stack pointer is fetched if the CPU is not executing on the interrupt stack.
3. The old processor status longword is written onto the new stack.
4. If the event being processed is either an interrupt between instructions or a trap, this step is not performed. A copy of the program counter is stored. Then the program counter is written onto a new stack. The value that is saved on the stack points to an event or the next instruction to execute.
5. The new processor status longword is initialized.
6. The interrupt priority level is changed only if
 - the event is an interrupt.
 - the event is an exception and the processor status longword vector bits 0 and 1 is a value of 1.
7. Any and all related parameters are stored.

8. For exceptions only, the *previous mode* field of the processor status longword is set to the old value of the current mode.
9. Last, the program counter is changed to point to the longword bits 2 through 31 of the vector.

If the processor received an *access control violation* or a *translation not valid* condition while attempting to write information on the kernel stack, a *kernel stack not valid—abort* is initiated. And the interrupt priority level is changed to 1F (hexadecimal). Any additional information associated with the original exception is lost. However, the processor status longword and the program counter are written to the interrupt stack with the same values as would have been written on the kernel stack. If the processor receives an *access control violation* or a *translation not valid* condition while attempting to write to the interrupt stack, the processor is halted and only the state of interrupt stack pointer, program counter, and processor status longword is ensured to be correct for subsequent analysis. The processor status longword and the program counter have the values that would have been written on the interrupt stack. The value of the processor status longword trace pending bit that is saved on the stack is shown in Table 8-4. The value of the program counter that is saved on the stack is shown in Table 8-5.

Table 8-4 ■ Trace Pending Bit Saved Values

Cause of Event	Source of Value Saved
Abort	PSL TP bit
BPT instruction	(Bit is cleared.)
CHM instruction	PSL TP bit
Fault	(Bit is cleared.)
Interrupt	If EPD bit is set, this bit is cleared. If after traps, before trace—from PSL TP bit.
Trace	(Bit is cleared.)
Trap	PSL TP bit
Reserved instructions	(Bit is cleared.)
XEC instruction	(Bit is cleared.)

Table 8-5 ■ Program Counter Saved Values

Cause of Event	Interrupt Stack Points to
Abort	The instruction aborting or detecting the <i>kernel-stack-not-valid</i> condition (not ensured on a <i>machine check</i> event).
BPT instruction	The BPT instruction.
CHM instruction	The next instruction to execute.
Fault	The instruction faulting.
Interrupt	The instruction interrupted or the next instruction to execute.
Trace	The next instruction to execute; that is, the instruction at the beginning of which the trace fault was taken.
Trap	The next instruction to execute.
Reserved instruction	The reserved instruction.
XFC instruction	The XFC instruction.

The noninterrupt stack pointers may be fetched and stored by hardware in either privileged registers or in the processor control block. Only LDPCTX and SVPCTX instructions always fetch the processor control block. *Move from processor register* and *move to processor register* instructions always fetch and store the pointers whether in privileged registers or the processor control block.

THE UNIVERSITY OF CHICAGO

PHYSICS DEPARTMENT

RESEARCH REPORT

NO. 1234

BY J. D. JARVIS

Chapter 9 ■ The Instruction Set

The instructions are arranged in alphabetic order. Notation conventions, instruction format, and addressing modes and conventions are described in detail in Chapter 5. No attempt is made to reiterate those details in this chapter. A general description of the instructions by category is in Chapter 6.

■ Add

Purpose: Used to perform arithmetic addition

Format: There are two formats—two operand and three operand.

operator *add.rx, sum.mx*

operator *add1.rx, add2.rx, sum.wx*

Opcode	Operator	Function
80	ADDB2	Add Byte 2 Operand
81	ADDB3	Add Byte 3 Operand
A0	ADDW2	Add Word 2 Operand
A1	ADDW3	Add Word 3 Operand
C0	ADDL2	Add Longword 2 Operand
C1	ADDL3	Add Longword 3 Operand
40	ADDF2	Add F__ floating 2 Operand
41	ADDF3	Add F__ floating 3 Operand
60	ADDD2	Add D__ floating 2 Operand
61	ADDD3	Add D__ floating 3 Operand
40FD	ADDG2	Add G__ floating 2 Operand
41FD	ADDG3	Add G__ floating 3 Operand
60FD	ADDH2	Add H__ floating 2 Operand
61FD	ADDH3	Add H__ floating 3 Operand

Description: In 2-operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3-operand format, the addend1 operand is added to the addend2 operand and the sum operand is replaced by the result. In floating-point format, the result is rounded.

■ Add Aligned Word Interlocked

Purpose: Used to maintain operating system resource usage counts

Format: ADAWI *add.rw, sum.mw*

Opcode	Operator	Function
58	ADAWI	Add Aligned Word Interlocked

Description: The addend operand is added to the sum operand and the sum operand is replaced by the result. The operation is interlocked against similar operations by other processors or in a multiple multiprocessor system. The destination must be aligned on a word boundary. Otherwise a reserved operand fault is taken.

NOTE

If the addend and the sum operand overlap, the result and the condition codes are *unpredictable*.

■ Add Compare and Branch

Purpose: Used to maintain a loop count and loop

Format: *operator limit.rx, add.rx, index.mx, displ.bw*

Opcode	Operator	Function
9D	ACBB	Add Compare and Branch Byte
3D	ACBW	Add Compare and Branch Word
F1	ACBL	Add Compare and Branch Longword
4F	ACBF	Add Compare and Branch F__ floating
6F	ACBD	Add Compare and Branch D__ floating
4FFD	ACBG	Add Compare and Branch G__ floating
6FFD	ACBH	Add Compare and Branch H__ floating

Description: The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal, or if the addend is negative and the comparison is greater than or equal, then the sign-extended branch displacement is added to the program counter (PC) and the PC is replaced by the result.

NOTE

ACB efficiently implements the general FOR or DO loops in high-level languages because the sense of the comparison between index and limit is dependent on the sign of the addend.

▪ Add One and Branch

Purpose: Used to increment an integer loop count and loop

Format: *operator limit.rl, index.ml, displ.bb*

Opcode	Operator	Function
F2	AOBLSS	Add One and Branch Less Than
F3	AOBLEQ	Add One and Branch Less Than or Equal

Description: One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. On AOBLSS, if the index operand is less than the limit operand, the branch is taken. On AOBLEQ, if the index operand is less than or equal to the limit operand, the branch is taken. If the branch is taken, the sign-extended branch displacement is added to the program counter (PC) and the PC is replaced by the result.

▪ Add Packed

Purpose: Used to add one packed decimal string to another

Format: There are two formats—4 operand and 6 operand.

ADDP4 *addlen.rw, addadr.ab, sumlen.rw, sumadr.ab*

ADDP6 *add1len.rw, add1adr.ab, add2len.rw, add2adr.ab, sumlen.rw, sumadr.ab*

Opcode	Operator	Function
20	ADDP4	Add Packed 4 Operand
21	ADDP6	Add Packed 6 Operand

Description: In 4-operand format, the addend string specified by the addend length and addend address operands is added to the sum string specified by the sum length and sum address operands and the sum string is replaced by the result.

In 6-operand format, the addend1 string specified by the addend1 length and addend1 address operands is added to the addend2 string specified by the addend2 length and addend2 address operands. The sum string specified by the sum length and sum address operands is replaced by the result.

▪ Add with Carry

Purpose: Used to perform extended-precision addition

Format: ADWC *add.rl, sum.ml*

Opcode	Mnemonic	Function
D8	ADWC	Add with Carry

Description: The contents of the condition code C bit and the addend operand are added to the sum operand. The sum operand is replaced by the result.

▪ Arithmetic Shift

Purpose: Used to shift integers

Format: *operator count.rb, source.rx, destination.wx*

Opcode	Operator	Function
78	ASHL	Arithmetic Shift Longword
79	ASHQ	Arithmetic Shift Quadword

Description: The source operand is arithmetically shifted by the number of bits specified by the count operand, and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing zeros into the least significant bit. A negative count operand shifts to the right bringing in copies of the most significant (sign) bit into the most significant bit position. A zero count operand replaces the destination operand with the unshifted source operand.

▪ Arithmetic Shift and Round Packed

Purpose: Used to scale numeric content of a packed decimal string by a power of 10

Format:

ASHP *cnt.rb, srclen.rw, srcadr.ab, round.rb, dstlen.rw, dstadr.ab*

Opcode	Operator	Function
F8	ASHP	Arithmetic Shift and Round Packed

Description: The source string specified by the source length and source address operands is scaled by a power of 10 specified by the count operand. The destination string specified by the destination length and destination address operands is replaced by the result.

A positive count operand effectively multiplies. A negative count effectively divides. A zero count just moves and affects condition codes. When a negative count is specified, the result is rounded using the round operand.

▪ Bit Clear

Purpose: Used to perform complemented AND of two integers

Format: There are two formats—2 operand and 3 operand

operator mask.rx, destination.mx

operator mask.rx, source.rx, destination.wx

Opcode	Operator	Function
8A	BICB2	Bit Clear Byte 2 Operand
8B	BICB3	Bit Clear Byte 3 Operand
AA	BICW2	Bit Clear Word 2 Operand
AB	BICW3	Bit Clear Word 3 Operand
CA	BICL2	Bit Clear Longword 2 Operand
CB	BICL3	Bit Clear Longword 3 Operand

Description: In 2-operand format, the destination operand is ANDed with the one's complement of the mask operand and the destination operand is replaced by the result. In 3-operand format, the source operand is ANDed with the one's complement of the mask operand and the destination operand is replaced by the result.

▪ Bit Clear Processor Status Longword

Purpose: Used to clear trap enables

Format: BICPSW *mask.rw*

Opcode	Operator	Function
B9	BICPSW	Bit Clear PSW

Description: On BICPSW, the Processor Status Longword is ANDed with the one's complement of the 16-bit mask operand and the PSW is replaced by the result.

▪ Bit Set

Purpose: Used to perform logical inclusive OR of two integers

Format: There are two formats—2 operand and 3 operand

operator mask.rx, destination.mrx

operator mask.rx, source.rx, destination.wx

Opcode	Operator	Function
88	BISB2	Bit Set Byte 2 Operand
89	BISB3	Bit Set Byte 3 Operand
A8	BISW2	Bit Set Word 2 Operand
A9	BISW3	Bit Set Word 3 Operand
C8	BISL2	Bit Set Longword 2 Operand
C9	BISL3	Bit Set Longword 3 Operand

Description: In 2-operand format, the mask operand is ORed with the destination operand and the destination operand is replaced by the result. In 3-operand format, the mask operand is ORed with the source operand and the destination operand is replaced by the result.

▪ Bit Set Processor Status Longword

Purpose: Used to set trap enables

Format: BISPSW *mask.rw*

Opcode	Operator	Function
B8	BISPSW	Bit set PSW

Description: On BISPSW, the Processor Status Longword is ORed with the 16-bit mask operand and the PSW is replaced by the result.

▪ Bit Test

Purpose: Used to test a set of bits for all zero

Format: *operand mask.rx, source.rx*

Opcode	Operator	Function
93	BITB	Bit Test Byte
B3	BITW	Bit Test Word
D3	BITL	Bit Test Longword

Description: The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.

▪ Branch

Purpose: Used to transfer control

Format: *operator* *displ.bx*

Opcode	Operator	Function
11	BRB	Branch with Byte Displacement
31	BRW	Branch with Word Displacement

Description: The sign-extended branch displacement is added to the program counter (PC) and the PC is replaced by the result.

▪ Branch on Bit

Purpose: Used to test a selected bit

Format: *operator* *pos.rl, base.vb, displ.bb*

Opcode	Operator	Function
E0	BBS	Branch on Bit Set
E1	BBC	Branch on Bit Clear

Description: The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC) and PC is replaced by the result.

▪ Branch on Bit Interlocked

Purpose: Used to test and modify a specified bit under memory interlock

Format: *operator* *pos.rl, base.vb, displ.bb*

Opcode	Operator	Function
E6	BBSSI	Branch on Bit Set and Set Interlocked
E7	BBCCI	Branch on Bit Clear and Clear Interlocked

Description: The single bit field specified by the *pos* and *base* operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC) and PC is replaced by the result. Regardless of whether or not the branch is affected, the tested bit is put in the new state as indicated by the instruction. If the bit is stored in memory, the reading of the state of the bit and the setting of it to the new state constitute an interlocked operation, interlocked against similar operations by other processors or devices in the system.

▪ Branch on Bit and Modify without Interlock

Purpose: Used to test and modify a specified bit

Format: *operator pos.rl, base.vb, displ.bb*

Opcode	Operator	Function
E2	BBSS	Branch on Bit Set and Set
E3	BBCS	Branch on Bit Clear and Set
E4	BBSC	Branch on Bit Set and Clear
E5	BBCC	Branch on Bit Clear and Clear

Description: The single bit field specified by the position (*pos*) and *base* operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the program counter (PC) and PC is replaced by the result. Regardless of whether or not the branch is taken, the tested bit is put in the new state as indicated by the instruction.

▪ Branch on Condition

Purpose: Used to test condition codes

Format: *operator displ.bb*

†	Condition	Operator	Function
12	Z EQL 0	BNEQ	Branch on Not Equal (Signed)
12	Z EQL 0	BNEQU	Branch on Not Equal Unsigned
13	Z EQL 1	BEQL	Branch on Equal (Signed)
13	Z EQL 1	BEQLU	Branch on Equal Unsigned
14	{N OR Z} EQL 0	BGTR	Branch on Greater Than (Signed)
15	{N OR Z} EQL 1	BLEQ	Branch on Less Than or Equal (Signed)
18	N EQL 0	BGEQ	Branch on Greater Than or Equal (Signed)
19	N EQL 1	BLSS	Branch on Less Than (Signed)
1A	{C OR Z} EQL 0	BGTRU	Branch on Greater Than Unsigned
1B	{C OR Z} EQL 1	BLEQU	Branch Less Than or Equal Unsigned
1C	V EQL 0	BVC	Branch on Overflow Clear
1D	V EQL 1	BVS	Branch on Overflow Set
1E	C EQL 0	BGEQU	Branch on Greater Than or Equal Unsigned
1E	C EQL 0	BCC	Branch on Carry Clear
1F	C EQL 1	BLSSU	Branch on Less Than Unsigned
1F	C EQL 1	BCS	Branch on Carry Set

† Opcode

Description: The condition codes are tested, and if the *condition* indicated by the instruction is met, the sign-extended branch displacement is added to the program counter (PC) and PC is replaced by the result.

The VAX conditional branch instructions permit considerable flexibility in branching but you need to exercise some care to choose the correct one. The conditional branch instructions are divided into three overlapping groups:

1. The Overflow and Carry Group

BVSV	EQL 1
BVCV	EQL 0
BCSC	EQL 1
BCCC	EQL 0

These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes.

2. The Unsigned Group

BLSSU	C EQL 1
BLEQU	{C or Z} EQL 1
BEQLU	Z EQL 1
BNEQU	Z EQL 0
BGEQU	C EQL 0
BGTRU	{C OR Z} EQL 0

These instructions typically follow integer and field instructions where the operands are treated as unsigned integers, addressed instructions, and character string instructions.

3. The Signed Group

BLSS	N EQL 1
BLEQ	{N OR Z} EQL 1
BEQL	Z EQL 1
BNEQ	Z EQL 0
BGEQ	N EQL 0
BGTR	{N OR Z} EQL 0

These instructions typically follow integer and field instructions where the operands are being treated as signed integers, floating-point instructions, and decimal-string instructions.

▪ Branch on Low Bit

Purpose: Used to test a specified bit

Format: *operator source.rl, displacement.bb*

Opcode	Operator	Function
E8	BLBS	Branch on Low Bit Set
E9	BLBC	Branch on Low Bit Clear

Description: The low bit (bit 0) of the source operand is tested and if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to program counter (PC) and PC is replaced by the result.

▪ Branch to Subroutine

Purpose: Used to transfer control to subroutine

Format: *operator displ.bx*

Opcode	Operator	Function
10	BSBB	Branch to Subroutine with Byte Displacement
30	BSBW	Branch to Subroutine with Word Displacement

Description: The program counter (PC) is pushed on the stack as a longword. The sign-extended branch displacement is added to PC and PC is replaced by the result.

NOTE

Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls, with the stack used for linkage. The form of such a call is JSB @(SP) + .

▪ Breakpoint Fault

Purpose: Used to help to implement debugging

Format: BPT

Opcode	Operator	Function
03	BPT	Breakpoint Fault

Description: This instruction is used with the trace bit of the processor status word to implement debugging facilities.

▪ Bugcheck

Purpose: Used to report software-detected errors

Format: *operator message.bx*

Opcode	Operator	Function
FEFF	BUGW	Bugcheck with Word Message Identifier
FDFD	BUGL	Bugcheck with Longword Message Identifier

Description: The hardware treats these opcodes as *Reserved to Digital* and *faults*. The VAX/VMS operating system treats these as requests to report software-detected errors. The inline message identifier is zero-extended to a longword (BUGW) and interpreted as a condition value. If the process is privileged to report bugs, a log entry is made. If the process is not privileged, a reserved instruction is signaled.

▪ Call Procedure with General Argument List

Purpose: Used to invoke a procedure with actual arguments from anywhere in memory

Format: CALLG *arglist.ab, dst.ab*

Opcode	Operator	Function
FA	CALLG	Call Procedure with General Argument List

Description: The stack pointer (SP) is saved in a temporary register and then bits 1:0 are replaced by zero so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0. The contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords, along with the program counter, frame pointer, and argument pointer. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a zero in bit 29 and bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and in bits 15 through 0 of the processor status word, with the T bit cleared is pushed on the stack. A longword zero is pushed on the stack. The frame pointer is replaced by the stack pointer. The argument pointer is replaced by the *arglist* operand that specifies the address of the actual argument list. The trap enables in the processor status word are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared. The T bit is unaffected. PC is replaced by the sum of destination operand and 2 that transfers control to the called procedure at the byte beyond the entry mask.

NOTE

The VMS Procedure Calling Software Standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. Registers 2 through 11 which are modified in the called procedure must be preserved in the mask.

▪ Call Procedure with Stack Argument List

Purpose: Used to invoke a procedure with actual arguments or addresses on the stack

Format: CALLS *numarg.rl, dst.ab*

Opcode	Operator	Function
--------	----------	----------

FB	CALLS	Call Procedure with Stack Argument List
----	-------	---

Description: The *numarg* operand is pushed on the stack as a longword (byte 0 contains the number of arguments; the high-order 24 bits are used by Digital software). SP is saved in a temporary location and then bits 1:0 of SP are replaced by zero so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of the register whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 1 in bit 29, 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with the T bit cleared is pushed on the stack. A longword zero is pushed on the stack. FP is replaced by SP. AP is set to the saved SP (the value of the Stack Pointer after the number of arguments operand was pushed on the stack). The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared; the T bit is unaffected. PC is replaced by the sum of destination operand and 2, which transfers control to the called procedure at the byte beyond the entry mask.

NOTES

1. Normally, the arglist is pushed onto the stack in reverse order prior to the CALLS. On return, the arglist is removed from the stack automatically.
2. The VMS Procedure Calling Software Standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers 2 through 11 that are modified in the called procedure must be preserved in the entry mask.

■ Case

Purpose: Used to perform multiple branching depending upon arithmetic input

Format:

operator selector.rx, base.rx, limit.rx, displ [0].bw, ..., displ [limit].bw

Opcode	Operator	Function
8F	CASEB	Case Byte
AF	CASEW	Case Word
CF	CASEL	Case Longword

Description: The base operand is subtracted from the selector operand and a temporary operand is replaced by the result. The temporary operand is compared with the limit operand and if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to the program counter (PC) and the PC is replaced by the result. Otherwise, 2 times the sum of the limit operand and 1 is added to the PC and the PC is replaced by the result. This causes the PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.

This instruction implements high-level language computed GOTO statements. You supply a list of displacements that generate different branch addresses depending on the value you obtain as a selector. The branch falls through if the selector does not generate any of the displacements on the list.

NOTE

After operand evaluation, PC is pointing at displ[0]—not the next instruction. The branch displacements are relative to the address of displ[0].

■ Change Mode

Purpose: Used to request higher privilege software

Format: *operator code.rw*

Opcode	Operator	Function
BC	CHMK	Change Mode to Kernel
BD	CHME	Change Mode to Executive
BE	CHMS	Change Mode to Supervisor
BF	CHMU	Change Mode to User

Description: Change mode instructions allow processors to change their access mode in a controlled manner. The instruction increases privilege only. A change in mode also results in a change of stack pointers. The old pointer is saved, and the new pointer is loaded. The PSL, PC, and any code passed by the instruction are pushed onto the stack of the new mode. The saved PC addresses the instruction following the CHMx instruction. The code is sign extended. After execution, the new stack's appearance is

sign extended code : (SP)

PC of next instruction

Old PSL

The destination mode selected by the opcode is used to select a location from the System Control Block. This location addresses the CHMx dispatcher for the specified mode.

▪ Clear

Purpose: Used to clear a scalar quantity

Format: *operator destination.wx*

Opcode	Operator	Function
94	CLRB	Clear Byte
B4	CLRW	Clear Word
D4	CLRL	Clear Longword
7C	CLRQ	Clear Quadword
7CFD	CLRO	Clear Octaword
7C	CLRD	Clear D__ floating
D4	CLRF	Clear F__ floating
7C	CLRG	Clear G__ floating
7CFD	CLRH	Clear H__ floating

Description: The destination operand is replaced by 0.

▪ Compare

Purpose: Used to perform an arithmetic comparison between two specified scalar quantities

Format: *operator src1.rx, src2.rx*

Opcode	Operator	Function
91	CMPB	Compare Byte
B1	CMPW	Compare Word
D1	CMPL	Compare Longword
51	CMPF	Compare F__ floating
71	CMPD	Compare D__ floating
51FD	CMPG	Compare G__ floating
71FD	CMPH	Compare H__ floating

Description: The src1 operand is compared with the src2 operand. The only action is to affect the condition codes.

▪ Compare Characters

Purpose: Used to compare two character strings

Format: There are two formats—3 operand and 5 operand.

CMPC3 *len.rw, src1adr.ab, src2adr.ab*

CMPC5 *src1len.rw, src1adr.ab, fill.rb, src2len.rw, src2adr.ab*

Opcode	Operator	Function
29	CMPC3	Compare Characters 3 Operand
2D	CMPC5	Compare Characters 5 Operand

Description: In 3-operand format, the first string is specified by the src1adr operand. The second string is specified by the src2adr operand. The strings are compared until inequality is detected or until all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison.

In 5-operand format, the bytes of one string are compared to the bytes of the second string. If one string is longer than the other, the shorter string is conceptually extended to the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison.

NOTE

1. After execution of a 3-operand instruction R0 = number of bytes remaining in string 1 including byte that terminated comparison, R1 = address of the byte in string 1 that terminated comparison, R2 = R0, R3 = address of the byte in string 2 that terminated the comparison.

2. After execution of a 5-character instruction R0 = the number of bytes remaining in string 1 including the byte that terminated comparison, R1 = address of the byte in string 1 that terminated comparison, R2 = the number of bytes remaining in string 2 including the byte that terminated comparison, and R3 = the address of the byte in string 2 that terminated comparison.

▪ Compare Field

Purpose: Used to compare bit field to integer

Format: *operator pos.rl, size.rb, base.vb, src.rl*

Opcode	Operator	Function
EC	CMPV	Compare Field
ED	CMPZV	Compare Zero-extended Field

Description: The field specified by the position, size, and base operands is compared with the source operand. For CMPV, the source operand is compared with the sign-extended field. For CMPZV, the source operand is compared with the zero-extended field. The only action is to affect the condition codes.

▪ Compare Packed

Purpose: Used to compare two packed decimal strings and set condition codes

Format: There are two formats—3 operand and 4 operand.

CMPP *len.rw, src1adr.ab, src2adr.ab*

CMPP *src1len.rw, src1adr.ab, src2len.rw, src2adr.ab*

Opcode	Operator	Function
35	CMPP3	Compare Packed—3 Operand
37	CMPP4	Compare Packed—4 Operand

Description: In 3-operand format, the src1 string is compared to the src2 string. The only action is to affect the condition codes.

In 4-operand format, the src1 string is compared to the src2 string. The only action is to affect the condition codes.

▪ Convert

Purpose: Used to convert a signed quantity to a different signed data type

Format: *operator* ... *src.rx*, *dst.wy*

Opcode	Operator	Function
99	CVTBW	Convert Byte to Word
98	CVTBL	Convert Byte to Longword
33	CVTWB	Convert Word to Byte
32	CVTWL	Convert Word to Longword
F6	CVTLB	Convert Longword to Byte
F7	CVTLW	Convert Longword to Word
4C	CVTBF	Convert Byte to F__ floating
6C	CVTBD	Convert Byte to D__ floating
4CFD	CVTBG	Convert Byte to G__ floating
6CFD	CVTBH	Convert Byte to H__ floating
4D	CVTWF	Convert Word to F__ floating
6D	CVTWD	Convert Word to D__ floating
4DFD	CVTWG	Convert Word to G__ floating
6DFD	CVTWH	Convert Word to H__ floating
4E	CVTLF	Convert Longword to F__ floating
6E	CVTLD	Convert Longword to D__ floating
4EFD	CVTLG	Convert Longword to G__ floating
6EFD	CVTLH	Convert Longword to H__ floating
48	CVTFB	Convert F__ floating to Byte
68	CVTDB	Convert D__ floating to Byte
48FD	CVTGB	Convert G__ floating to Byte
68FD	CVTHB	Convert H__ floating to Byte
49	CVTFW	Convert F__ floating to Word
69	CVTDW	Convert D__ floating to Word
49FD	CVTGW	Convert G__ floating to Word
69FD	CVTHW	Convert H__ floating to Word

4A	CVTFL	Convert F__ floating to Longword
4B	CVTRFL	Convert Rounded F__ floating to Longword
6A	CVTDL	Convert D__ floating to Longword
6B	CVTRDL	Convert Rounded D__ floating to Longword
4AFD	CVTGL	Convert G__ floating to Longword
48FD	CVTRGL	Convert Rounded G__ floating to Longword
6AFD	CVTHL	Convert H__ floating to Longword
6BFD	CVTRHL	Convert Rounded H__ floating to Longword
56	CVTFD	Convert F__ floating to D__ floating
99FD	CVTFG	Convert F__ floating to G__ floating
98FD	CVTFH	Convert F__ floating to H__ floating
76	CVTDF	Convert D__ floating to F__ floating
32FD	CVTDH	Convert D__ floating to H__ floating
33FD	CVTGF	Convert G__ floating to F__ floating
56FD	CVTGH	Convert G__ floating to H__ floating
F6FD	CVTHF	Convert H__ floating to F__ floating
F7FD	CVTHD	Convert H__ floating to D__ floating
76FD	CVTHG	Convert H__ floating to G__ floating

Description: The source operand is converted to the data type of the destination operand and the destination operand is replaced by the result. For integer format, conversion of a shorter data type to a longer is done by sign extension. Conversion of longer to a shorter is done by truncation of the higher numbered (most significant) bits. For floating format, the form of the conversion is as follows:

Exact Conversion	Truncated Conversion	Rounded Conversion
CVTBF	CVTHW	CVTLF
CVTBD	CVTFL	CVTDF
CVTBG	CVTFB	CVTRHL
CVTBH	CVTDL	CVTRFL
CVTWF	CVTDB	CVTRDL
CVTWD	CVTGL	CVTHG
CVTWG	CVTHL	CVTRGL
CVTWH	CVTGB	CVTGF
CVTLD	CVTFW	CVTHF
CVTFD	CVTDW	CVTHD
CVTLG	CVTGW	
CVTLH	CVTHB	
CVTFH		
CVTFG		
CVTDH		
CVTGH		

▪ Convert Leading Separate Numeric to Packed

Purpose: Used to convert leading separate numeric string to packed decimal string

Format: CVTSP *srcLen.rw, srcadr.ab, dstLen.rw, dstadr.ab*

Opcode	Operator	Function
09	CVTSP	Convert Leading Separate Numeric to Packed

Description: The source numeric string specified by the source length and source address operands is converted to a packed decimal string, and the destination string specified by the destination address and destination length operands is replaced by the result.

▪ Convert Longword to Packed

Purpose: Used to convert longword integer to packed decimal string

Format: CVTLP *src.rl, dstLen.rw, dstadr.ab*

Opcode	Operator	Function
F9	CVTLP	Convert Long to Packed

Description: The destination string is specified by the destination length and address operands. The source operand is converted to a packed decimal string and the destination string is replaced by the result.

▪ Convert Packed to Leading Separate Numeric

Purpose: Used to convert packed decimal string to leading separate numeric string

Format: CVTPS *srcLen.rw, srcadr.ab, dstLen.rw, dstadr.ab*

Opcode	Operator	Function
08	CVTPS	Convert Packed to Leading Separate Numeric

Description: The source packed decimal string specified by the source length and source address operands is converted to a leading separate numeric string. The destination string specified by the destination length and destination address operands is replaced by the result.

Conversion is effected by replacing the lowest addressed byte of the destination string by the ASCII plus or minus characters which is determined by the sign of the source string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

▪ Convert Packed to Longword

Purpose: Used to convert a packed decimal string to a longword

Format: CVTPL *srcLen.rw, srcadr.ab, dst.wl*

Opcode	Operator	Function
36	CVTPL	Convert Packed to Long

Description: The source string specified by the source length and source address operands is converted to a longword and the destination operand is replaced by the result.

▪ Convert Packed to Trailing Numeric

Purpose: Used to convert packed decimal string to trailing numeric string

Format: CVTPT *srcLEN.rw, srcadr.ab, tbladr.ab, dstLEN.rw, dstadr.ab*

Opcode	Operator	Function
--------	----------	----------

24	CVTPT	Convert Packed to Trailing Numeric
----	-------	------------------------------------

Description: The source packed decimal string specified by the source length and source address operands is converted to a trailing numeric string. The destination string specified by the destination length and destination address operands is replaced by the result. The condition code N and Z bits are affected by the value of the source packed decimal string.

Conversion is effected by using the highest addressed byte of the source string as an unsigned index into a 256-byte table whose zeroth entry address is specified by the table address operand. The byte read out of the table replaces the least significant byte of the destination string. The remaining bytes of the destination string are replaced by the ASCII representations of the values of the corresponding packed decimal digits of the source string.

NOTE

By appropriate specification of the table, conversion to any form of trailing numeric string may be realized. See Chapter 4 for the preferred form of trailing overpunch, zoned, and unsigned data. In addition, the table may be set up for absolute value, negative absolute value, or negative conversions.

▪ Convert Trailing Numeric to Packed

Purpose: Used to convert trailing numeric string to packed decimal string

Format: CVTTP *srclen.rw, srcadr.ab, tbladr.ab, dstlen.rw, dstadr.ab*

Opcode	Operator	Function
26	CVTTP	Convert Trailing Numeric to Packed

Description: The source trailing numeric string specified by the source length and source address operands is converted to a packed decimal string. The destination packed decimal string specified by the destination address and destination length operands is replaced by the result.

Conversion is effected by using the highest addressed (trailing) byte of the source string as an unsigned index into a 256-byte table whose zeroth entry is specified by the table address operand. The byte read out of the table replaces the highest addressed byte of the destination string; that is, the byte containing the sign and the least significant digit. The remaining packed digits of the destination string are replaced by the low-order four bits of the corresponding bytes in the source string.

NOTES

1. By appropriate specification of the table, conversion from any form of trailing numeric string may be realized. In addition, the table may be set up for absolute value, negative absolute value or negated conversions. Refer to Chapter 4 for the preferred form of trailing overpunch, zoned, and unsigned data.
2. If the table translation produces a sign nibble containing any valid sign, the preferred sign representation is stored in the destination packed decimal string.

▪ Cyclic Redundancy Check Instruction

Purpose: Used to initiate communications or software redundancy checks

Format: CRC *tbl.ab, inicrc.rl, strlen.rw, stream.ab*

Opcode	Operator	Function
0B	CRC	Calculate Cyclic Redundancy Check

Description: The CRC of the data stream described by the string descriptor is calculated. The initial CRC is given by *inirc* and is normally 0 or -1 unless the CRC is calculated in several steps. R0 is replaced by the result. If the polynomial is less than order-32, the result must be extracted from R0. The CRC polynomial is expressed by the contents of the 16-longword table. See the Notes for calculation of the table.

NOTES

1. If the data stream is not a multiple of eight bits long, it must be right-adjusted with leading zero fill.
2. If the CRC polynomial is less than order-32, the result must be extracted from the low-order bits of R0.
3. The following algorithm can be used to calculate the CRC table given a polynomial expressed as follows:

poly < n > — {coefficient of $x^{**\{\text{order}-1-n\}}$ }

This routine is available as system library routine LIB\$-CRC__TABLE (POLY, TABLE). The table is the location of a 64-byte (16-longword) table into which the result is written.

4. The following are descriptions of some commonly used CRC polynomials.

▪ CRC-16 (used in DDCMP and Bisync):

Polynomial: $X^{16} + X^{15} + X^2 + 1$

Poly: 120001 (octal)

Initialize: 0

Result: R0 < 15:0 >

▪ CCITT (used in ADCCP, HDLC, SDLC):

Polynomial: $X^{16} + X^{12} + X^5 + 1$

Poly: 102010 (octal)

Initialize: -1 < 15:0 >

Result: one's complement of R0 < 15:0 >

▪ AUTODIN-II

Polynomial: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$

Poly: EDB88320 (hexadecimal)

Initialize: $-1 < 31:0 >$

Result: one's complement of $R0 < 31:0 >$

5. This instruction produces an *unpredictable* result unless the table is well formed, such as produced in Note 3. Note that for any well-formed table, entry [0] is always 0 and entry [8] is always the polynomial expressed as in Note 3. The operation can be implemented using shifts of one, two, or four bits at a time as follows:

Shift: 1

Steps per byte (limit): 8

Index table index: $\text{tmp3} < 0 >$

Table multiplier: 8

Use table entries: $[0] = 0, < 8 >$

Shift: 2

Steps per byte (limit): 4

Index table index: $\text{tmp3} < 1:0 >$

Table multiplier: 4

Use table entries: $[0] = 0, [4], [8], [12]$

Shift: 4

Steps per byte (limit): 2

Index table index: $\text{tmp3} < 3:0 >$

Table multiplier: 1

Use table entries: all

■ Decrement

Purpose: Used to subtract 1 from an integer

Format: *operator difference.mx*

Opcode	Operator	Function
97	DECB	Decrement Byte
B7	DECW	Decrement Word
D7	DECL	Decrement Longword

Description: One is subtracted from the difference operand and the difference operand is replaced by the result.

■ Divide

Purpose: Used to perform arithmetic division

Format: There are two formats—2 operand and 3 operand.

operator divr.rx, quo.mx

operator divr.rx, divd.rx, quo.wx

Opcode	Operator	Function
86	DIVB2	Divide Byte 2 Operand
87	DIVB3	Divide Byte 3 Operand
A6	DIVW2	Divide Word 2 Operand
A7	DIVW3	Divide Word 3 Operand
C6	DIVL2	Divide Longword 2 Operand
C7	DIVL3	Divide Longword 3 Operand
46	DIVF2	Divide F__ floating 2 Operand
47	DIVF3	Divide F__ floating 3 Operand
66	DIVD2	Divide D__ floating 2 Operand
67	DIVD3	Divide D__ floating 3 Operand
46FD	DIVG2	Divide G__ floating 2 Operand
47FD	DIVG3	Divide G__ floating 3 Operand
66FD	DIVH2	Divide H__ floating 2 Operand
67FD	DIVH3	Divide H__ floating 3 Operand

Description: In 2-operand format, the quotient operand is divided by the divisor operand, and the quotient operand is replaced by the result.

In 3-operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the result. In floating format, the quotient operand result is rounded for both 2- and 3-operand formats.

Integer division is performed so that the remainder (unless it is zero) has the same sign as the dividend. That is, the result is truncated toward zero.

▪ Divide Packed

Purpose: Used to divide one packed decimal string by a second, with the result placed in a third

Format:

DIVP *divrlen.rw, divradr.ab, divdlen.rw, divdadr.ab, quolen.rw, quoadr.ab*

Opcode	Operator	Function
27	DIVP	Divide Packed

Description: The dividend string is specified by the dividend length and dividend address operands. The divisor string is specified by the divisor length and divisor address operands. The quotient string is specified by the quotient length and quotient address operands. The dividend string is divided by the divisor string. The quotient string is replaced by the result. The division is performed in the following manner:

1. The absolute value of the remainder (which is lost) is less than the absolute value of the divisor.
2. The product of the absolute value of the quotient and the absolute value of the divisor is less than or equal to the absolute value of the dividend.
3. The sign of the quotient is determined by the rules of algebra from the signs of the dividend and the divisor. If the value of the quotient is zero, the sign is always positive.

▪ Edit Instruction

Purpose: Used to edit a source string

Format: EDITPC *srclen.rw, srcadr.ab, pattern.ab, dstadr.ab*

Opcode	Operator	Function
38	EDITPC	Edit Packed to Character String

Description: The destination string is specified by the pattern and destination address (dstadr) operands. The source string is specified by the source length (srclen) and source address (srcadr) operands. The destination string is replaced by the edited version of the source string.

Editing is performed according to the pattern string. The pattern string consists of one-byte pattern operators. Editing starts at the address pattern and extends until an end (EO\$END) pattern operator is encountered. Some pattern operators take no operands. Some take a repeat count that is in the rightmost nibble of the pattern operator itself. The rest take a one-byte operand that immediately follows the pattern operator. This operand is either an unsigned integer length or a byte character. Pattern operators are described on subsequent pages and are summarized in Table 9-1.

Table 9-1 • Edit Instruction Pattern Operators

Function	Name	Operand*	Summary
Control:	EO\$ADJUST__ INPUT	<i>len</i>	Adjust source length.
	EO\$CLEAR__ SIGNIF	—	Clear significance flag.
	EO\$END	—	End edit.
	EO\$SET__ SIGNIF	—	Set significance flag.
Fixup:	EO\$BLANK__ ZERO	<i>len</i>	Fill backward when zero.
	EO\$REPLACE__ SIGN	<i>len</i>	Replace with fill if -0.
Insert:	EO\$FILL	<i>rep</i>	Insert fill.
	EO\$INSERT	<i>char</i>	Insert character, fill if insignificant.
	EO\$STORE__ SIGN	—	Insert sign.
Load:	EO\$LOAD__ FILL	<i>char</i>	Load fill character.
	EO\$LOAD__ MINUS	<i>char</i>	Load sign character if negative.
	EO\$LOAD__ PLUS	<i>char</i>	Load sign character if positive.
	EO\$LOAD__ SIGN	<i>char</i>	Load sign character.
Move:	EO\$END__ FLOAT	—	End floating sign.
	EO\$FLOAT	<i>rep</i>	Move digits, floating sign.
	EO\$MOVE	<i>rep</i>	Move digits, filling insignificant.

* *char* = one character

len = length in the range 1 through 255

rep = repeat counter in the range 1 through 15

The following definitions are used:

fill = R2 < 7:0 >

sign = R2 < 15:8 >

EO\$ADJUST__ INPUT

Purpose: Used to handle source strings of a length different from the output string

Format: EO\$ADJUST__ INPUT *len*

Opcode	Pattern Operator	Function
47	EO\$ADJUST__ INPUT	Adjust Input Length

Description: The pattern operator is followed by an unsigned byte integer length in the range 1 through 31. If the source string has more digits than the length, the excess digits are read and discarded. If any discarded digits are not zero, the overflow and significance bits are set, and the zero bit is cleared. If the source string has fewer digits than the length, a counter is set to the number of leading zeros to supply. This counter is stored as a negative number in register R0 in bits 31 through 16.

EO\$BLANK__ ZERO

Purpose: Used to fix the destination to be blank when the source value is zero

Format: EO\$BLANK__ ZERO *len*

Opcode	Pattern Operator	Function
45	EO\$BLANK__ ZERO	Blank Backwards When Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero, the contents of the fill register are stored into the last length bytes of the destination string.

NOTE

This pattern operator is used to blank any characters stored in the destination under a forced significance such as a sign or the digits following the radix point.

EO\$CLEAR__ SIGNIF

Purpose: Used to control the significance (leading zero) indicator

Format: EO\$CLEAR__ SIGNIF

Opcode	Pattern Operator	Function
02	EO\$CLEAR__ SIGNIF	Clear Significance

Description: The significance indicator is cleared. This controls the treatment of leading zeros. (Leading zeros are zero digits for which the significance indicator is clear.) EO\$CLEAR__ SIGNIF is used to initialize leading zero suppression (EO\$MOVE) or floating sign (EO\$FLOAT) following a fixed insert (EO\$INSERT with significance set).

EO\$END

Purpose: Used to end edit operation

Format: EO\$END

Opcode	Pattern Operator	Function
00	EO\$END	End Edit

Description: The edit operation is terminated.

EO\$END__ FLOAT

Purpose: Used to end a floating-sign operation

Format: EO\$END__ FLOAT

Opcode	Pattern Operator	Function
01	EO\$END__ FLOAT	End Floating Sign

Description: If the floating sign has not yet been placed in the destination (that is, if significance is not set), the contents of the sign register are stored in the destination and significance is set.

NOTE

This pattern operator is used after a sequence of one or more EO\$FLOAT pattern operators that start with significance clear. The EO\$FLOAT sequence can include intermixed EO\$INSERTs and EO\$FILLs.

EO\$FILL

Purpose: Used to insert the fill character

Format: EO\$FILL *rep*

Opcode	Pattern Operator	Function
81:8F	EO\$FILL	Store Fill

Description: The right nibble of the pattern operator is the repeat count. The contents of the fill register are placed into the destination the number of times specified in the *rep* operand. This pattern operator is used for fill (blank) insertion.

EO\$FLOAT

Purpose: Used to move digits, floating the sign across insignificant digits

Format: EO\$FLOAT *rep*

Opcode	Pattern Operator	Function
A1:AF	EO\$FLOAT	Float Sign

Description: The right nibble of the pattern operator is the repeat count. For repeat iterations, the following algorithm is executed the number of times specified in the repeat count (*rep*) operand.

Repeat Count Algorithm—The next digit from the source is examined for one of two conditions:

-
- If the next digit is nonzero and significance is not yet set, the contents of the sign register are stored in the destination, the significance bit is set, and the zero bit is cleared.
-
- If the digit is significant, it is stored in the destination. Otherwise, the content of the fill register is stored in the destination.
-

This pattern operator is used to move digits with a floating arithmetic sign. The sign must already be set up as for EO\$STORE__ SIGN. A sequence of one or more EO\$FLOATs can include intermixed EO\$INSERTs and EO\$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO\$END__ FLOAT.

This pattern operator is used to move digits with a floating currency sign. The sign must already be set up with an EO\$LOAD__ SIGN. A sequence of one or more EO\$FLOATs can include intermixed EO\$INSERTs and EO\$FILLs. Significance must be clear before the first pattern operator of the sequence. The sequence must be terminated by one EO\$END__ FLOAT.

EO\$INSERT

Purpose: Used to insert a fixed character, substituting the fill character if not significant

Format: EO\$INSERT *char*

Opcode	Pattern Operator	Function
44	EO\$INSERT	Insert Character

Description: The pattern operator is followed by a character. If the significance bit is set, the character is placed into the destination. If the significance bit is not set, the contents of the fill register are placed into the destination.

NOTE

This pattern operator is used for blankable inserts (for example, comma) and fixed inserts (for example, slash). Fixed inserts require that significance be set by EO\$SET__ SIGNIF or EO\$END__ FLOAT.

EO\$LOAD

Purpose: Used to change the contents of the fill or sign register

Format: *pattern__operator char*

Opcode	Pattern Operator	Function
40	EO\$LOAD__ FILL	Load Fill Register
41	EO\$LOAD__ SIGN	Load Sign Register
42	EO\$LOAD__ PLUS	Load Sign Register If Plus
43	EO\$LOAD__ MINUS	Load Sign Register If Minus

Description: The pattern operator is followed by a character. For EO\$LOAD__ FILL, this character is placed into the fill register. For EO\$LOAD__ SIGN, this character is placed into the sign register. For EO\$LOAD__ PLUS, this character is placed into the sign register if the source string has a positive sign. For EO\$LOAD__ MINUS, this character is placed into the sign register if the source string has a negative sign.

NOTES

1. EO\$LOAD__ FILL is used to set up check protection instead of space.
2. EO\$LOAD__ SIGN is used to set up a floating currency sign.
3. EO\$LOAD__ PLUS is used to set up a nonblank plus sign.
4. EO\$LOAD__ MINUS is used to set up a alternate minus sign such as CR, DB, or the PL/1 + .

EO\$MOVE

Purpose: Used to move digits, filling for insignificant digits (leading zeros)

Format: EO\$MOVE *rep*

Opcode	Pattern Operator	Function
91:9F	EO\$MOVE	Move Digits

Description: The right nibble of the pattern operator is the repeat count. For repeat iterations, the following algorithm is executed the number of times specified in the repeat count (rep) operand.

The next digit is moved from the source to the destination under the following conditions:

- If the digit is nonzero, the significance bit is set and the zero bit is cleared.
- If the digit is not significant (that is, it is a leading zero), it is replaced by the contents of the fill register in the destination.

NOTES

1. This pattern operator is used to move digits without a floating sign. If leading zero suppression is desired, the significance bit must be clear. If leading zero should be explicit, the significance bit must be set. A string of EO\$MOVE operators intermixed with EO\$INSERT and EO\$FILL operators correctly handles suppression.
2. If check protection (*) is desired, EO\$LOAD__ FILL must precede the EO\$MOVE.

EO\$REPLACE__ SIGN

Purpose: Used to change the destination sign when the value is minus zero

Format: EO\$REPLACE__ SIGN *len*

Opcode	Pattern Operator	Function
46	EO\$REPLACE__ SIGN	Replace Sign When Minus Zero

Description: The pattern operator is followed by an unsigned byte integer length. If the value of the source string is zero (that is, if the Z bit is set), the contents of the fill register are stored into the byte of the destination string that is *len* bytes before the current position.

NOTES

1. The length must be nonzero and within the destination string already produced.
2. This pattern operator is used to correct a stored sign (EO\$END__ FLOAT or EO\$STORE__ SIGN) if a minus was stored and the source value is zero.

EO\$SET__ SIGNIF

Purpose: Used to control the significance (leading zero) indicator

Format: EO\$SET__ SIGNIF

Opcode	Pattern Operator	Function
03	EO\$SET__ SIGNIF	Set Significance

Description: The significance indicator is set. This controls the treatment of leading zeros. (Leading zeros are zero digits for which the significance indicator is clear.) EO\$SET__ SIGNIF is used to avoid leading suppression (before EO\$MOVE) or to force a fixed insert (before EO\$INSERT).

EO\$STORE__ SIGN

Purpose: Used to insert the sign character

Format: EO\$STORE__ SIGN

Opcode	Pattern Operator	Function
04	EO\$STORE__ SIGN	Store Sign

Description: The contents of the sign register are placed into the destination.

NOTE

This pattern operator is used for any nonfloating arithmetic sign. It should be preceded by a EO\$LOAD__ PLUS and/or EO\$LOAD__ MINUS if the default sign convention is not desired.

▪ Exclusive OR

Purpose: Used to perform logical exclusive OR of two integers

Format: There are two formats—2 operand and 3 operand

operator mask.rx, destination.mx

operator mask.rx, source.rx, destination.wx

Opcode	Operator	Function
8C	XORB2	Exclusive OR Byte 2 Operand
8D	XORB3	Exclusive OR Byte 3 Operand
AC	XORW2	Exclusive OR Word 2 Operand
AD	XORW3	Exclusive OR Word 3 Operand
CC	XORL2	Exclusive OR Longword 2 Operand
CD	XORL3	Exclusive OR Longword 3 Operand

Description: In 2-operand format, the mask operand is XORed with the destination operand and the destination operand is replaced by the result. In 3-operand format, the mask operand is XORed with the source operand and the destination operand is replaced by the result.

▪ Extended Divide

Purpose: Used to perform extended-precision division

Format: EDIV *divisor.rl, dividend.rq, quotient.wl, remainder.wl*

Opcode	Operator	Function
7B	EDIV	Extended Divide

Description: The dividend operand is divided by the divisor operand. The quotient operand is replaced by the quotient and the remainder operand is replaced by the remainder.

Unless the remainder operand is zero, the division is performed so that the remainder operand has the same sign as the dividend operand. If the quotient and remainder operands both reference the same location, the remainder operand overlays the quotient operand.

▪ Extended Function Call

Purpose: Used to provide customer-defined extensions to the instruction set.

Format: XFC

Opcode	Operator	Function
FC	XFC	Extended Function Call

Description: This instruction requests services of nonstandard microcode or software. If no special microcode is loaded, then an exception is generated to a kernel mode software simulator. Typically, the next byte would specify which of several extended functions are requested. Parameters would be passed either as normal operands or, more likely, in fixed registers.

▪ Extended Modulus

Purpose: Used to perform accurate range reduction of math function arguments

Format: There are four formats—one for each type of floating point data

EMODD *mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx*

EMODF *mulr.rx, mulrx.rb, muld.rx, int.wl, fract.wx*

EMODG *mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx*

EMODH *mulr.rx, mulrx.rw, muld.rx, int.wl, fract.wx*

Opcode	Operator	Function
54	EMODF	Extended Multiply and Integerize F__ floating
74	EMODD	Extended Multiply and Integerize D__ floating
54FD	EMODG	Extended Multiply and Integerize G__ floating
74FD	EMODH	Extended Multiply and Integerize H__ floating

Description: The multiplier extension operand is concatenated with the multiplier operand to gain 8 (EMODD and EMODF), 11 (EMODG), or 15 (EMODH) additional low-order fraction bits. The low-order 5 or 1 bits of the 16-bit multiplier extension operand are ignored by the EMODG and EMODH instructions, respectively. The multiplicand operand is multiplied by the extended multiplier operand. This multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in F__ floating, 64 bits in D__ floating and G__ floating, and 128 in H__ floating. Regarding the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.

■ Extended Multiply

Purpose: Used to perform extended-precision multiplication

Format: EMUL *multiplier.rl, multiplicand.rl, addend.rl, product.wq*

Opcode	Operator	Function
7A	EMUL	Extended

Description: The multiplicand operand is multiplied by the multiplier operand giving a double-length result. The addend operand is sign-extended to double length and added to the result. Then, the product operand is replaced by the result.

■ Extract Field

Purpose: Used to move bit field to integer

Format: *operator pos.rl, size.rb, base.vb, dst.wl*

Opcode	Operator	Function
EE	EXTV	Extract Field
EF	EXTZV	Extract Zero-extended Field

Description: For EXTV, the destination operand is replaced by the sign-extended field specified by the position, size, and base operands. For EXTZV, the destination operand is replaced by the zero-extended field specified by the position, size, and base operands. If the size operand is zero, the only action is to replace the destination operand with zero and affect the condition codes.

An example of this instruction is to extract the four protection bits (bits 27 through 30) from the memory management unit Page Table Entry. The base address is the address of a longword operand containing these bits; the position operand could be the number of bits from the base address to the protection code; and the size operand would be 4 because the protection code is 4 bits long. The destination operand would specify where the protection bits are to be stored.

Because the protection code is not an arithmetic operand and does not need to be sign-extended, the *extract zero-extended field* instruction should be specified.

▪ Find First Bit

Purpose: Used to locate the first bit in a bit field

Format: *operator startpos.rl, size.rb, base.vb, findpos.wl*

Opcode	Operator	Function
EB	FFC	Find First Clear
EA	FFS	Find First Set

Description: A field specified by the start position, size, and base operands is extracted. The field is tested for a bit in the state indicated by the instruction starting at bit 0 and extending to the highest bit in the field. If a bit in the indicated state is found, the find position operand is replaced by the position of the bit and the Z condition code bit is cleared. If no bit in the indicated state is found, the find position operand is replaced by the position (relative to the base) of a bit one position to the left of the specified field and the Z condition code bit is set. If the size operand is zero, the find position operand is replaced by the start position operand and the Z condition code bit is set.

▪ Halt

Purpose: Used to stop processor operation

Format: HALT

Opcode	Operator	Function
00	HALT	Halt

Description: If the process is running in kernel mode, the processor is halted. Otherwise, a privileged instruction fault occurs.

NOTE

This opcode is zero to trap many branches to data.

▪ Increment

Purpose: Used to add 1 to an integer

Format: *operator sum.mx*

Opcode	Operator	Function
96	INCB	Increment Byte
B6	INCW	Increment Word
D6	INCL	Increment Longword

Description: One is added to the sum operand and the sum operand is replaced by the result.

▪ Index

Purpose: Used for index calculation of arrays of fixed length data, bit fields, and strings

Format:

INDEX *subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl*

Opcode	Operator	Function
0A	INDEX	Compute Index

Description: The index in operand is added to the subscript operand and the sum is multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a subscript range trap is taken.

▪ Insert Entry in Queue

Purpose: Used to add an entry to the head or tail of a queue

Format: INSQUE *entry.ab, predecessor.ab*

Opcode	Operator	Function
0E	INSQUE	Insert Entry in Queue

Description: The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, it is cleared. The insertion is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

NOTES

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQUE instruction is implemented so the cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or tail of the queue.
3. During access validation, any access that cannot be completed results in a memory management exception, even though the queue insertion is not started.

▪ Insert Entry into Queue at Head, Interlocked

Purpose: Used to perform an interlocked entry insert at head of queue

Format: INSQHI *entry.ab, header.aq*

Opcode	Operator	Function
5C	INSQHI	Insert Entry into Queue at Head, Interlocked

Description: The entry specified by the entry operand is inserted into the queue following the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise, it is cleared. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a processor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

NOTES

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQHI instruction is implemented so the cooperating software processes in a processor may access a shared list without additional synchronization.
3. During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion is not started.

■ **Insert Entry into Queue at Tail, Interlocked**

Purpose: Used to perform an interlocked entry insert at tail of queue

Format: INSQTI *entry.ab, header.aq*

Opcode Operator Function

5D INSQTI Insert Entry into Queue at Tail, Interlocked

Description: The entry specified by the entry operand is inserted into the queue preceding the header. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a processor environment. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates.

NOTES

1. Because the insertion is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The INSQTI instruction is implemented so the cooperating software processes in a processor may access a shared list without additional synchronization.
3. During access validation, any access that cannot be completed results in a memory management exception even though the queue insertion was not started.

▪ Insert Field

Purpose: Used to move an integer to a bit field

Format: INSV *source.ri, position.ri, size.rb, base.vb*

Opcode	Operator	Function
--------	----------	----------

F0	INSV	Insert Field
----	------	--------------

Description: The field specified by the position, size, and base operands is replaced by bits $\langle \text{size} - 1:0 \rangle$ of the source operand. If the size operand is zero, the only action is to affect the condition codes.

▪ Jump

Purpose: Used to transfer control

Format: JMP *dst.ab*

Opcode	Operator	Function
--------	----------	----------

17	JMP	Jump
----	-----	------

Description: The PC is replaced by the destination operand.

▪ Jump to Subroutine

Purpose: Used to transfer control to subroutine

Format: JSB *dst.ab*

Opcode	Operator	Function
--------	----------	----------

16	JSB	Jump to Subroutine
----	-----	--------------------

Description: The program counter (PC) is pushed on the stack as a longword. The PC is replaced by the destination operand.

NOTE

Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for coroutine calls, with the stack used for linkage. The form of such a call is: JSB @(SP) + .

■ Load Process Context

Purpose: Used to restore register and memory management context

Format: LDPCTX

Opcode	Operator	Function
06	LDPCTX	Load Process Context

Description: The process control block is specified by the process control block base. The general registers are loaded from the PCB. The memory management registers describing the process address space are also loaded and the process entries in the translation buffer are cleared. Execution is switched to the kernel stack. The PC and PSL are moved between the PCB and the stack suitable for use by a subsequent REI instruction. This instruction can be executed only in kernel mode.

Some processors keep a copy of each of the process stack pointers in internal registers. In those processors, LDPCTX loads the internal registers from the PCB. Processors that do not keep a copy of all four process stack pointers in internal registers keep only the current access mode register in an internal register. The contents of the internal register are switched with the PCB contents whenever the current access mode field changes.

■ Locate Character

Purpose: Used to find a character in a character string

Format: LOCC *char.rb, len.rw, adr.ab*

Opcode	Operator	Function
3A	LOCC	Locate Character

Description: The character (char) operand is compared with the bytes of the string specified by the length (len) and address (adr) operands. Comparison continues until equality is detected, or until all bytes of the string have been compared. If equality is detected, the condition code Z bit is cleared. Otherwise the Z bit is set.

▪ Match Characters

Purpose: Used to find substring (object) in character string

Format: MATCHC *objlen.rw, objadr.ab, srclen.rw, srcadr.ab*

Opcode	Operator	Function
39	MATCHC	Match Characters

Description: The source string is specified by the source length and source address operands. The object string is specified by the object length and object address operands. The source string is examined for a substring that matches the object string. If the substring is found, the condition code Z bit is set. Otherwise, it is cleared.

▪ Move

Purpose: Used to move a specified scalar quantity

Format: *operator source.rx, destination.wx*

Opcode	Operator	Function
90	MOVB	Move Byte
B0	MOVW	Move Word
D0	MOVL	Move Longword
7D	MOVQ	Move Quadword
7DFD	MOVO	Move Octaword
50	MOVF	Move F__ floating
70	MOVD	Move D__ floating
50FD	MOVG	Move G__ floating
70FD	MOVH	Move H__ floating

Description: The destination operand is replaced by the source operand. The source operand is unaffected.

NOTE

The MOV_B and MOV_W instructions do not modify the high-order bytes of a register destination. Refer to the MOV_{ZxL} and CVT_{xL} instructions to update the full register contents.

■ Move Address

Purpose: Calculates address of quantity

Format: *operator source.ax, destination.wl*

Opcode	Operator	Function
9E	MOVAB	Move Address Byte
3E	MOVAW	Move Address Word
DE	MOVAL	Move Address Longword
DE	MOVAF	Move Address F__ floating
7E	MOVAQ	Move Address Quadword
7E	MOVAD	Move Address D__ floating
7E	MOVAG	Move Address G__ floating
7EFD	MOVAH	Move Address H__ floating
7EFD	MOVAO	Move Address Octaword

Description: The destination operand is replaced by the source operand, which is an address. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

■ Move Characters

Purpose: Used to move a character string or block of memory

Formats: There are two formats—3 operand and 5 operand

MOV_C *src_{len}.rw, src_{adr}.ab, dst_{adr}.ab*

MOV_C *src_{len}.rw, src_{adr}.ab, fill.rb, dst_{len}.rw, dst_{adr}.ab*

Opcode	Operator	Function
28	MOV _{C3}	Move Character—3 Operand
2C	MOV _{C5}	Move Character—5 Operand

Description: The destination string is replaced by the source string. If the destination string is longer than the source string, the highest address bytes of the destination are replaced by the fill operand. However, if the destination string is shorter than the source string, the highest addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.

▪ Move Complement

Purpose: Used to move the logical complement of an integer

Format: *operator source.rx destination.wx*

Opcode	Operator	Function
92	MCOMB	Move Complement Byte
B2	MCOMW	Move Complement Word
D2	MCOML	Move Complement Longword

Description: The destination operand is replaced by the one's complement of the source operand.

▪ Move from Processor Register

Purpose: Used to provide access to the internal privileged (processor) registers

Format: MFPR *procreg.rl, dst.wl*

Opcode	Operator	Function
DB	MFPR	Move from Processor Register

Description: The specified register is stored. The *procreg* operand is a longword that contains the privileged register number. Execution may have register-specific side effects. A reserved operand fault may occur if the processor internal register does not exist. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode. (See also "Move to Processor Register".) See Table 9-2 for a list of the processor registers.

Table 9-2 • Processor (Privileged) Registers

Number	Register Name	Mnemonic*	Type†	Scope‡
0	Kernel Stack Pointer	KSP	R/W	PROC
1	Executive Stack Pointer	ESP	R/W	PROC
2	Supervisor Stack Pointer	SSP	R/W	PROC
3	User Stack Pointer	USP	R/W	PROC
4	Interrupt Stack Pointer	ISP	R/W	CPU
8	P0 Base Register	POBR	R/W	PROC
9	P0 Length Register	POLR	R/W	PROC
10	P1 Base Register	P1BR	R/W	PROC
11	P1 Length Register	P1LR	R/W	PROC
12	System Base Register	SBR	R/W	CPU
13	System Length Register	SLR	R/W	CPU
16	Process Control Block Base	PCBB	R/W	PROC
17	System Control Block Base	SCBB	R/W	CPU
18	Interrupt Priority Level	IPL	R/W	CPU
19	Asynchronous System Trap Level	ASTLVL	R/W	PROC
20	Software Interrupt Request	SIRR	W	CPU
21	Software Interrupt Summary	SISR	R/W	CPU

* Each register address is formed as PR\$ followed by the register's mnemonic. For example, the register address for the user stack pointer is PR\$USP. Once assigned, the register number is not changed. Implementation-dependent registers are assigned distinct addresses for each implementation. Thus, any privileged register present on more than one implementation performs the same function whenever implemented. All unsigned positive numbers are reserved to Digital. All negative numbers are reserved to Digital's Customer Software Services and customers.

† The Type column indicates the read/write characteristics of that register. The letter R means the register is read-only. The characters R/W means the register is both read and write. The character W means the register is write-only.

‡ The Scope column indicates if a register is a CPU register or a process register. Registers labeled CPU are manipulated through software only using the MTPR and MFPR instructions. Registers labeled PROC are manipulated by context switch instructions.

Table 9-2 ■ Processor (Privileged) Registers (Cont.)

Number	Register Name	Mnemonic*	Type†	Scope‡
24	Interval Clock Control	ICCS	R/W	CPU
25	Next Interval Count	NICR	W	CPU
26	Interval Count	ICR	R	CPU
27	Time of Year (optional)	TODR	R/W	CPU
32	Console Receive Control/status	RXCS	R/W	CPU
33	Console Receiver Data Buffer	RXDB	R	CPU
34	Console Transmit Control/status	TXCS	R/W	CPU
35	Console Transmit Data Buffer	TXDB	W	CPU
56	Memory Management Enable	MAPEN	R/W	CPU
57	Translation Buffer Invalidate All	TBIA	W	CPU
58	Translation Buffer Invalidate Single	TBIS	W	CPU
61	Performance Monitor Enable	PMR	R/W	PROC
62	System Identification	SID	R	CPU

■ Move from Processor Status Longword

Purpose: Used to obtain processor status

Format: MOVPSL *dst.wl*

Opcode	Operator	Function
DC	MOVPSL	Move from PSL

Description: The destination operand is replaced by the processor status longword.

■ Move Negated

Purpose: Used to move the arithmetic negation of a scalar quantity

Format: *operator source.rx, destination.wx*

Opcode	Operator	Function
8E	MNEGB	Move Negated Byte
AE	MNEGW	Move Negated Word
CE	MNEGL	Move Negated Longword
52	MNEGF	Move Negated F__ floating
72	MNEGD	Move Negated D__ floating
52FD	MNEGG	Move Negated G__ floating
72FD	MNEGH	Move Negated H__ floating

Description: The destination operand is replaced by the negative of the source operand.

▪ Move Packed

Purpose: Used to move a packed decimal string from one memory location to another memory location

Format: MOVP *len.rw, srcadr.ab, dstadr.ab*

Opcode	Operator	Function
34	MOVP	Move Packed

Description: The destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands.

▪ Move to Processor Register

Purpose: Used to provide access to the internal privileged registers

Format: MTPR *src.rl, procreg.rl*

Opcode	Operator	Function
DA	MTPR	Move to Processor Register

Description: The specified register is loaded. The *procreg* operand is a longword that contains the privileged register number. Execution may have register-specific side effects. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode. A reserved operand fault may occur if the processor internal register does not exist. See "Move from Processor Register" for a summary of accessible privileged registers (Table 9-2).

▪ Move Translated Characters

Purpose: Used to move and translate character strings

Format:

MOVTC *srclen.rw, srcadr.ab, fill.rb, tbladr.ab, dstlen.rw, dstadr.ab*

Opcode	Operator	Function
--------	----------	----------

MOVTC		Move Translated Characters
-------	--	----------------------------

Description: The source string is translated and replaces the destination string. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose first entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. If the destination string is longer than the source string, the highest addressed bytes of the destination string are replaced by the fill operand. If the destination string is shorter than the source string, the highest addressed bytes of the source string are not translated and moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result. If the destination string overlaps the translation table, the destination string is *unpredictable*.

▪ Move Translated until Character

Purpose: Used to move and translate a character string and to handle escape codes

Format:

MOVTUC *srclen.rw, srcadr.ab, esc.rb, tbladr.ab, dstlen.rw, dstadr.ab*

Opcode	Operator	Function
--------	----------	----------

2F	MOVTUC	Move Translated until Character
----	--------	---------------------------------

Description: The specified source string is translated and replaces the destination string. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose first entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. Translation continues until a translated byte is equal to the escape byte or until the source string or destination string is exhausted. If translation is terminated because of escape, the condition code V bit is set. Otherwise, it is cleared. If the destination string overlaps the table, the results are *unpredictable*. If the source and destination strings overlap and their addresses are not identical, then the results are *unpredictable*. If the source and destination string addresses are identical, the translation is performed correctly.

▪ Move Zero-extended

Purpose: Used to convert an unsigned integer to a wider unsigned integer

Format: *operator source.rx, destination.wy*

Opcode	Operator	Function
9B	MOVZBW	Move Zero-Extended Byte to Word
9A	MOVZBL	Move Zero-Extended Byte to Longword
3C	MOVZWL	Move Zero-Extended Word to Longword

Description: For MOVZBW, bits 7:0 of the destination operand are replaced by the source operand; bits 15:8 are replaced by zero. For MOVZBL, bits 7:0 of the destination operand are replaced by the source operand; bits 31:8 are replaced by zero. For MOVZWL, bits 15:0 of the destination operand are replaced by the source operand; bits 31:16 are replaced by zero.

▪ Multiply

Purpose: Used to perform arithmetic multiplication

Format: There are two formats—2 operand and 3 operand.

operator multiplier.rx, product.mx

operator multiplier.rx, multiplicand.rx, product.wx

Opcode	Operator	Function
84	MULB2	Multiply Byte 2 Operand
85	MULB3	Multiply Byte 3 Operand
A4	MULW2	Multiply Word 2 Operand
A5	MULW3	Multiply Word 3 Operand
C4	MULL2	Multiply Longword 2 Operand
C5	MULL3	Multiply Longword 3 Operand
44	MULF2	Multiply F__ floating 2 Operand
45	MULF3	Multiply F__ floating 3 Operand
64	MULD2	Multiply D__ floating 2 Operand
65	MULD3	Multiply D__ floating 3 Operand
44FD	MULG2	Multiply G__ floating 2 Operand
45FD	MULG3	Multiply G__ floating 3 Operand
64FD	MULH2	Multiply H__ floating 2 Operand
65FD	MULH3	Multiply H__ floating 3 Operand

Description: In 2-operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the result.

In 3-operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the result. In floating format, the product operand result is rounded for both 2- and 3-operand formats.

▪ Multiply Packed

Purpose: Used to multiply one packed decimal string by a second, result placed in a third

Format:

MULP *mulrlen.rw, mulradr.ab, muldlen.rw, muldadr.ab, prodlen.rw, prodadr.ab*

Opcode	Operator	Function
25	MULP	Multiply Packed

Description: The multiplicand string is specified by the multiplicand length and multiplicand address operands. The multiplier string is specified by the multiplier length and multiplier address operands. The product string specified by the product length and product address operands. The multiplicand string is multiplied by the multiplier string. The product string is replaced by the result.

▪ Polynomial Evaluation

Purpose: Used for fast calculation of math functions

Format: *operator argument.rx, degree.rw, table address.ab*

Opcode	Operator	Function
55	POLYF	Polynomial Evaluation F__ floating
75	POLYD	Polynomial Evaluation D__ floating
55FD	POLYG	Polynomial Evaluation G__ floating
75FD	POLYH	Polynomial Evaluation H__ floating

Description: The table address operand points to a table of polynomial coefficients. The coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at increasing addresses. The data type of the coefficients is the same as the data type of the argument operand.

Evaluation is carried out by Horner's method, and the contents of R0 (R1'R0 for POLYD and POLYG, R3'R2'R1'R0 for POLYH) are replaced by the result. The result computed is

$$\text{result} = C[0] + X*(C[1] + X*(C[2] + \dots X*C[d]))$$

where d = degree and X = arg. The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation. POLYH requires four longwords on the stack to store arg in case the instruction is interrupted.

▪ Pop Registers

Purpose: Used to restore multiple registers from stack

Format: POPR *mask.rw*

Opcode	Operator	Function
BA	POPR	Pop Registers

Description: The contents of registers whose number corresponds to set bits in the mask operand are replaced by longwords popped from the stack. R[n] is replaced if $\text{mask} < n >$ is set. The mask is scanned from bit 0 to bit 14 and bit 15 is ignored.

▪ Probe Accessibility

Purpose: Used to verify that arguments can be accessed

Format: *operator mode.rb, len.rw, base.ab*

Opcode	Operator	Function
OC	PROBER	Probe Read Accessibility
OD	PROBEW	Probe Write Accessibility

Description: The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero-extended length. The bytes in between are not checked. System software must check all pages between the two end bytes if they are to be accessed.

The protection is checked against the larger of the modes specified in bits $<1:0>$ of the mode operand and the previous mode field of the PSL. Note that probing with a mode operand of zero is equivalent to probing the mode specified in PSL $<\text{previous-mode}>$.

NOTES

1. On the probe of a process virtual address, if the valid bit of the system page table entry is zero, then a *translation not valid fault* occurs. This allows for the demand paging of the process page tables.
2. On the probe of a process virtual address, if the protection field of the system page table entry indicates *no access*, then a status of *not-accessible* is given. One *no access* page table entry in the system map is equivalent to 128 no access page table entries in the process map.

▪ **Push Address**

Purpose: Calculates address of quantity

Format: *operator* *source.ax*

Opcode	Operator	Function
9F	PUSHAB	Push Address Byte
3F	PUSHAW	Push Address Word
DF	PUSHAL	Push Address Longword
DF	PUSHAF	Push Address F__ floating
7F	PUSHAQ	Push Address Quadword
7F	PUSHAD	Push Address D__ floating
7F	PUSHAG	Push Address G__ floating
7FFD	PUSHAH	Push Address H__ floating
7FFD	PUSHAO	Push Address Octaword

Description: The source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.

▪ Push Longword

Purpose: Used to push a longword source operand onto the stack pointer

Format: PUSHL *src.rl*

Opcode	Operator	Function
DD	PUSHL	Push Longword

Description: The longword source (*src*) operand is pushed onto the stack.

▪ Push Registers

Purpose: Used to save multiple registers on stack

Format: PUSHR *mask.rw*

Opcode	Operator	Function
BB	PUSHR	Push registers

Description: The contents of registers whose number corresponds to set bits in the mask operand are pushed on the stack as longwords. R[n] is pushed if mask < n > is set. The mask is scanned from bit 14 to bit 0, and bit 15 is ignored.

▪ Remove Entry from Queue

Purpose: Used to remove an entry from the head or tail of a queue

Format: REMQUE *entry.ab, address.wl*

Opcode	Operator	Function
0F	REMQUE	Remove Entry from Queue

Description: The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the address of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise, it is cleared. If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, it is cleared. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.

NOTES

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The REMQUE instruction is implemented so the cooperating software processes in a single processor may access a shared list without additional synchronization if insertions and removals are only at the head or tail of the queue.
3. During access validation, any access that cannot be completed results in a memory management exception, even though the queue removal is not started.

▪ Remove Entry from Queue at Head, Interlocked

Purpose: Used to perform an interlocked remove of an entry from the head of queue

Format: REMQHI *header.aq, address.wl*

Opcode	Operator	Function
5E	REMQHI	Remove Entry from Queue at Head, Interlocked

Description: The queue entry following the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there is nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise, it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a processor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

NOTES

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The REMQHI instruction is implemented so the cooperating software processes in a processor may access a shared list without additional synchronization.

3. During access validation, any access that cannot be completed results in a memory management exception even though the queue removal is not started.

▪ Remove Entry from Queue at Tail, Interlocked

Purpose: Used to perform an interlocked entry remove from the tail of a queue

Format: REMQTI *header.aq, address.wl*

Opcode	Operator	Function
5F	REMQTI	Remove Entry from Queue Tail, Interlocked

Description: The queue entry preceding the header is removed from the queue. The address operand is replaced by the address of the entry removed. If no entry was removed from the queue (because either there is nothing to remove or secondary interlock failed), the condition code V bit is set; otherwise, it is cleared. If the interlock succeeded and the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise, it is cleared. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process even in a processor environment. The removal is a noninterruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state. If the instruction fails to acquire the secondary interlock, the instruction sets condition codes and terminates without altering the queue.

NOTES

1. Because the removal is noninterruptible, processes running in kernel mode can share queues with interrupt service routines.
2. The REMQTI instruction is implemented so the cooperating software processes in a processor may access a shared list without additional synchronization.
3. During access validation, any access that cannot be completed results in a memory management exception even though the queue removal is not started.

▪ Return from Exception or Interrupt

Purpose: Used to exit from an exception or interrupt service routine and initiate a controlled return.

Format: REI

Opcode	Operator	Function
02	REI	Return from Exception or Interrupt

Description: A longword is *popped* from the current stack and held in a temporary PC. A second longword is *popped* from the current stack and held in a temporary PSL. Validity of the *popped* PSL is checked. The current stack pointer is *saved* and a new stack pointer is selected according to the new PSL CURRENT__ MODE and IS fields. The level of the highest-privilege AST is checked against the current access mode to see whether a pending AST can be delivered. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction lookahead in the processor is reinitialized.

The exception or interrupt service routine is responsible for restoring any registers saved and removing any parameters from the stack.

▪ Return from Procedure

Purpose: Used to transfer control from a procedure to the calling process

Format: RET

Opcode	Operator	Function
04	RET	Return from Procedure

Description: The stack pointer (SP) is replaced by the frame pointer (FP) plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved PSW in bits 15:0 is popped from the stack and saved in a temporary register. The program counter (PC), frame pointer (FP), and argument pointer (AP) are replaced by longwords popped from the stack. A register restore mask is formed from bits 27:16 of the temporary register. Scanning from bit 0 to bit 11 of the restore mask, the contents of registers whose number is indicated by set bits in the mask are replaced by longwords popped from the stack. SP is incremented by bits 31:30 of the temporary register. PSW is replaced by bits 15:0 of the temporary register. If bit 29 in the temporary register is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.

The VMS Procedure Calling Software Standard and condition handling facility assume that procedures that return a function value or a status code do so in R0 or R0 and R1.

▪ Return from Subroutine

Purpose: Used to return control from subroutine

Format: RSB

Opcode	Operator	Function
05	RSB	Return from Subroutine

Description: The program counter (PC) is replaced by a longword removed from the stack.

NOTE

RSB is used to return from subroutines called by the BSBB, BSBW, and JSB instructions.

▪ Rotate Longword

Purpose: Used to rotate integer

Format: ROTL *count.rb, source.rl, destination.wl*

Opcode	Operator	Function
9C	ROTL	Rotate Longword

Description: The source operand is rotated logically by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.

▪ Save Process Context

Purpose: Used to save register context

Format: SVPCTX

Opcode	Operator	Function
07	SVPCTX	Save Process Context

Description: The process control block (PCB) is specified by the privileged register process control block base (PCBB). The general registers are saved into the PCB. The PC and PSL currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when the IS is clear, then the IS is set, the interrupt stack pointer is activated, and the IPL is maximized with 1 because of the switch to the interrupt stack. This instruction can be executed only in kernel mode.

NOTES

1. The map, ASTLVL, and PME contents of the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.
2. Some processors keep a copy of each of the process stack pointers in internal registers. In those processors, SVPCTX stores the internal registers in the PCB. Processors that do not keep a copy of all four process stack pointers in internal registers keep only current access mode register in an internal register and switch this with the PCB contents whenever the current access mode field changes.
3. Between the SVPCTX instruction that saves state for one process and the LDPCTX that loads the state of another, the internal stack pointers may not be referenced by MFPR or MTPR instructions. This implies that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers.

▪ Scan Characters

Purpose: Used to find (scan) a set of characters in character string

Format: SCANC *len.rw, adr.ab, tableadr.ab, mask.rb*

Opcode	Operator	Function
2A	SCANC	Scan Characters

Description: The bytes of the string specified by the length and address operands are successively used to index into a 256-byte table whose entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is nonzero or until all the bytes of the string have been exhausted. If a nonzero AND result is detected, the condition code Z bit is cleared. Otherwise, the Z bit is set.

▪ Skip Character

Purpose: Used to skip a character in a character string

Format: SKPC *char.rb, len.rw, adr.ab*

Opcode	Operator	Function
3B	SKPC	Skip Character

Description: The character (char) operand is compared with the bytes of the string specified by the length (len) and address (adr) operands. Comparison continues until inequality is detected, or until all bytes of the string have been compared. If inequality is detected, the condition code Z bit is cleared. Otherwise the Z bit is set.

■ Span Characters

Purpose: Used to skip (span) a set of characters in character string

Format: SPANC *len.rw, adr.ab, tableadr.ab, mask.rb*

Opcode	Operator	Function
2B	SPANC	Span Characters

Description: The bytes of the string specified by the length and address operands are successively used to index into a 256-byte table whose entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is zero or until all the bytes of the string have been exhausted. If a zero result is detected, the condition code Z bit is cleared. Otherwise, the Z bit is set.

■ Subtract

Purpose: Used to perform arithmetic subtraction

Format: There are two formats—2 operand and 3 operand

operator subtrabend.rx, difference.mx

operator subtrabend.rx, minuend.rx, difference.wx

Opcode	Operator	Function
82	SUBB2	Subtract Byte 2 Operand
83	SUBB3	Subtract Byte 3 Operand
A2	SUBW2	Subtract Word 2 Operand
A3	SUBW3	Subtract Word 3 Operand
C2	SUBL2	Subtract Longword 2 Operand
C3	SUBL3	Subtract Longword 3 Operand
42	SUBF2	Subtract F__ floating 2 Operand
43	SUBF3	Subtract F__ floating 3 Operand
62	SUBD2	Subtract D__ floating 2 Operand
63	SUBD3	Subtract D__ floating 3 Operand
42FD	SUBG2	Subtract G__ floating 2 Operand
43FD	SUBG3	Subtract G__ floating 3 Operand
62FD	SUBH2	Subtract H__ floating 2 Operand
63FD	SUBH3	Subtract H__ floating 3 Operand

Description: In 2-operand format, the subtrahend operand is subtracted from the difference operand and the difference operand is replaced by the result.

In 3-operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the result. In floating format, the result is rounded.

▪ Subtract One and Branch

Purpose: Used to decrement an integer loop count and loop

Format: *operator index.ml, displ.bb*

Opcode	Operator	Function
F4	SOBGEQ	Subtract One and Branch Greater Than or Equal to Zero
F5	SOBGTR	Subtract One and Branch Greater Than Zero

Description: One is subtracted from the index operand and the index operand is replaced by the result. On SOBGEQ, if the index operand is greater than or equal to 0, the branch is taken. On SOBGTR, if the index operand is greater than 0, the branch is taken. If the branch is taken, the sign-extended branch displacement is added to the program counter (PC) and the PC is replaced by the result.

▪ Subtract Packed

Purpose: Used to subtract one packed decimal string from another

Format: There are two formats—a 4-operand and a 6-operand format.

SUBP4 *sublen.rw, subadr.ab, diflen.rw, difadr.ab*

SUBP6 *sublen.rw, subadr.ab, minlen.rw, minadr.ab, diflen.rw, difadr.ab*

Opcode	Operator	Function
22	SUBP4	Subtract Packed 4 Operand
23	SUBP6	Subtract Packed 6 Operand

Description: In 4-operand format, the subtrahend string is specified by subtrahend length and subtrahend address operands. The difference string is specified by the difference length and difference address operands. The subtrahend string is subtracted from the difference string and the difference string is replaced by the result.

In 6-operand format, the subtrahend string is specified by the subtrahend length and subtrahend address operands. The minuend string is specified by the minuend length and minuend address operands. The difference string is specified by the difference length and difference address operands. The subtrahend string is subtracted from the minuend string. The difference string is replaced by the result.

▪ Subtract with Carry

Purpose: Used to perform extended-precision subtraction

Format: SBWC *subtrahend.rl, difference.ml*

Opcode	Operator	Function
D9	SWBC	Subtract with Carry

Description: The subtrahend operand and the contents of the condition code C bit are subtracted from the difference operand and the difference operand is replaced by the result.

▪ Test

Purpose: Used to perform an arithmetic compare of a scalar to 0

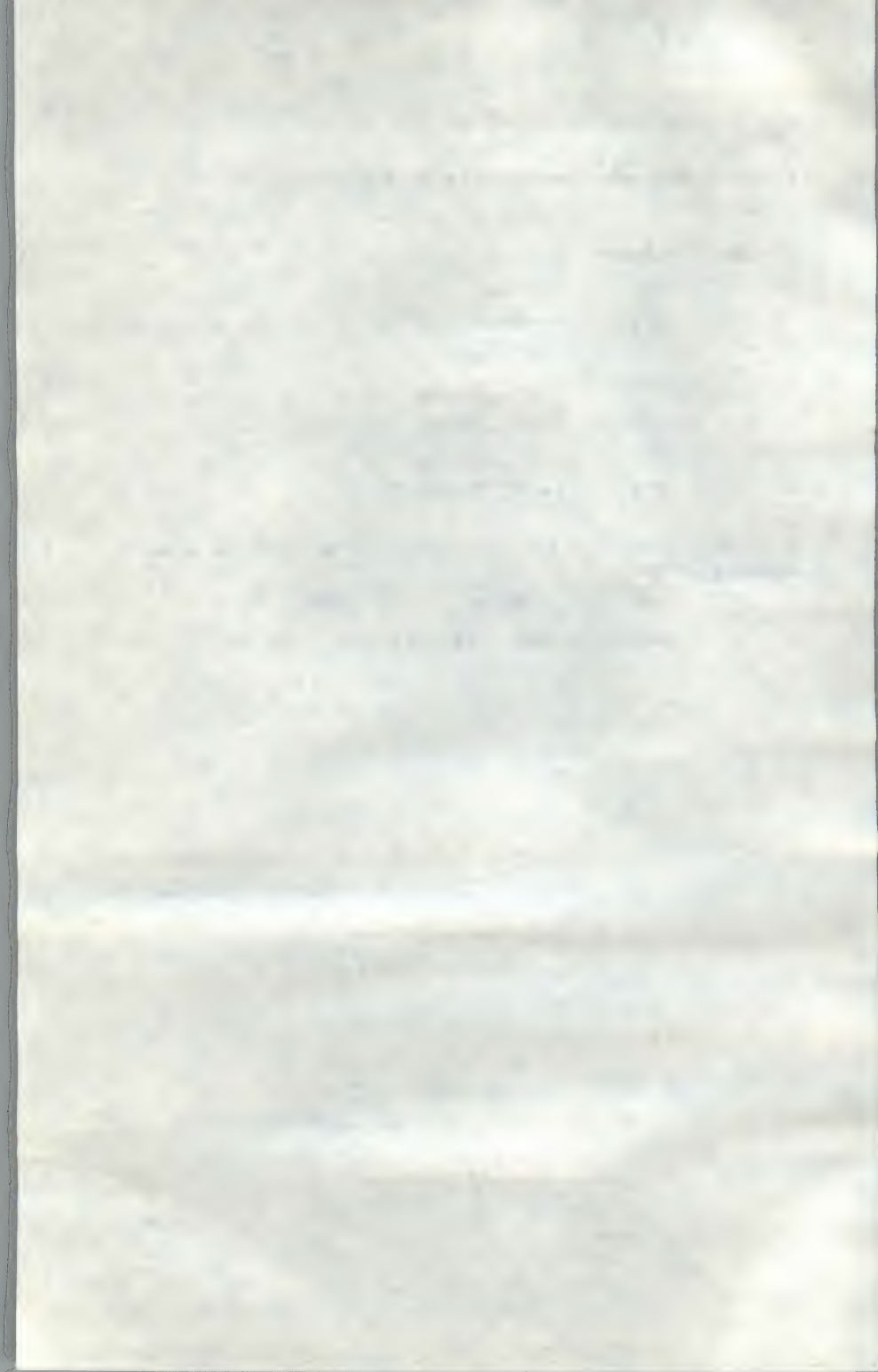
Format: *operator* *src.rx*

Opcode	Operator	Function
95	TSTB	Test Byte
B5	TSTW	Test Word
D5	TSTL	Test Longword
53	TSTF	Test F__ floating
73	TSTD	Test D__ floating
53FD	TSTG	Test G__ floating
73FD	TSTH	Test H__ floating

Description: The condition codes are affected according to the value of the source (src) operand.

NOTE

On a floating reserved operand, the condition codes are *unpredictable*.



Chapter 10 ■ Architectural Subsetting

Architectural subsetting deals with those parts of the VAX architecture that may be included as standard features of a processor, provided as options to the processor, or omitted completely from the processor.

A processor implementing a subset of the VAX instructions, data types, or registers, as described in this chapter, is known as a subset VAX processor. Of the many subsets possible, the following four subsets are the most common.

-
- Full VAX—includes all VAX data types, instructions, and registers.
 - Kernel subset—the minimum allowed subset.
 - MicroVAX I subset—as implemented by the MicroVAX I systems.
 - MicroVAX II subset—as implemented by the MicroVAX II chip.
-

■ Subsetting Rules

The features of the architecture that may be omitted are divided into several groups, each with different rules for subsetting. Floating and string instructions with their associated data types, compatibility mode instruction set, and processor registers may be omitted in a subset implementation.

Floating-point Instructions

The first group consists of the D__ floating, F__ floating, G__ floating, and H__ floating data types, and the associated instructions. Each of these data types may be subset only as an entity. This means that if one of these data types is included, all the instructions that operate on that data type must be included. If an instruction in this group is omitted by a processor, execution of the instruction results in a reserved instruction fault.

-
- D__ floating instructions (24)—ACBD, ADDD2, ADDD3, CMPD, CVTBD, CVTDB, CVTDF, CVTDL, CVTDW, CVTFD, CVTLD, CVTRDL, CVTWD, DIVD2, DIVD3, EMODD, MNEGD, MOVD, MULD2, MULD3, POLYD, SUBD2, SUBD3, and TSTD.
 - F__ floating instructions (22)—ACBF, ADDF2, ADDF3, CMPF, CVTBF, CVTFB, CVTFL, CVTFW, CVTLF, CVTRFL, CVTWF, DIVF2, DIVF3, EMOF, MNEGF, MOVF, MULF2, MULF3, POLYF, SUBF2, SUBF3, and TSTF.
-

-
- G__ floating instructions (24)—ACBG, ADDG2, ADDG3, CMPG, CVTBG, CVTFG, CVTGB, CVTGF, CVTGL, CVTGW, CVTLG, CVTRGL, CVTWG, DIVG2, DIVG3, EMODG, MNEGG, MOVG, MULG2, MULG3, POLYG, SUBG2, SUBG3, and TSTG.
-
- H__ floating instructions (32)—ACBH, ADDH2, ADDH3, CLRH (CLRO), CMPH, CVTBH, CVTDH, CVTFH, CVTGH, CVTHB, CVTHD, CVTHF, CVTHG, CVTHL, CVTHW, CVTLH, CVTRHL, CVTWH, DIVH2, DIVH3, EMODH, MNEGH, MOVAH (MOVAO), MOVH, MOVO, MULH2, MULH3, PUSHAH (PUSHAO), POLYH, SUBH2, SUBH3, and TSTH.
-

String Instructions

The second group consists of the string instructions and their associated data types, including the decimal string, EDITPC, CRC, and character string instructions, but not including MOV C3 or MOV C5 instructions. The MOV C3 and MOV C5 instructions are part of the kernel instruction set and may not be omitted. Instructions in this group may be subset individually.

-
- Character string instructions (10)—CMPC3, CMPC5, CRC, LOCC, MATCHC, MOVTC, MOVTUC, SCANC, SKPC, and SPANC.
-
- Decimal string instructions (17)—ADDP4, ADDP6, ASHP, CMPP3, CMPP4, CVTL P, CVTPL, CVTPT, CVTTP, CVTPS, CVTSP, DIVP, EDITPC, MOVP, MULP, SUBP4, and SUBP6.
-

If an instruction in this group is omitted, an attempt to execute the instruction results in a subset-emulation exception.

Compatibility Mode Instruction Set

The third group consists of the compatibility mode instruction set. If compatibility mode is omitted by a processor, the execution of an REI instruction attempting to enter compatibility mode results in a reserved operand fault.

Processor Registers

The fourth group consists of processor registers. The registers described below may be omitted from subset processors. If any of the registers named in one of the following subgroups is included, all the registers in that subgroup must be included.

-
- Interval timer registers NICR, ICR, ICCS except for <IE>. The ICCS<IE> register is part of the kernel subset and may not be omitted.
-

-
- Time-of-Year clock register TODR.
-
- Console registers RXCS, RXDB, TXCS, and TXDB.
-
- Performance Monitor Enable register PME.
-

▪ The Kernel Instruction Set

The kernel instruction set is defined by exception; it is those instructions that may not be omitted. For convenience, the kernel set is listed here. There are 304 native mode instructions in the full VAX instruction set. Of these, 129 may be omitted, leaving 175 instructions in the kernel instruction set. Byte, word, longword, and quadword operand sizes have been included in the kernel instruction set. The octaword operand size has not been included. The following instructions are the kernel instruction set.

-
- Address instructions (8)—MOVAB, MOVAL, MOVAQ, MOVAV, PUSHAB, PUSHAL, PUSHAQ, and PUSHAW.
-
- Branch and control instructions (39)—ACBB, ACBL, ACBW, AOBLEQ, AOBLSS, BBC, BBCC, BBCCI, BBBS, BBS, BBSC, BBSS, BBSSI, BEQL, BGEQ, BGEQU, BGTR, BGTRU, BLBC, BLBS, BNEQ, BRB, BRW, BSBB, BSBW, BVC, BVS, CASEB, CASEL, CASEW, JMP, JSB, RSB, SOBGEQ, and SOBGTR.
-
- Character string instructions (2)—MOV C3 and MOV C5.
-
- Instructions for use by operating systems (12)—CHME, CHMK, CHMS, CHMU, HALT, LDPCTX, MFPR, MTPR, PROBER, PROBEW, REI, and SVPCTX.
-
- Integer arithmetic and logical instructions (89)—ADAWI, AADB2, AADB3, ADDL2, ADDL3, ADDW2, ADDW3, ADWC, ASHL, ASHQ, BICB2, BICB3, BICL2, BICL3, BICW2, BICW3, BISB2, BISB3, BISL2, BISL3, BISW2, BISW3, BITB, BITL, BITW, CLRB, CLRL, CLRQ, CLRW, CMPB, CMPL, CMPW, CVTBL, CVTBW, CVTLB, CVTLW, CVTWB, CVTWL, DECB, DECL, DECW, DIVB2, DIVB3, DIVL2, DIVL3, DIVW2, DIVW3, EDIV, EMUL, INCB, INCL, INCW, MCOMB, MCOML, MCOMW, MNEGB, MNEGL, MNEGW, MOV B, MOVL, MOVQ, MOVW, MOVZBL, MOVZBW, MOVZWL, MULB2, MULB3, MULL2, MULL3, MULW2, MULW3, PUSHL, ROTL, SBWC, SUBB2, SUBB3, SUBL2, SUBL3, SUBW2, SUBW3, TSTB, TSTL, TSTW, XORB2, XORB3, XORL2, XORL3, XORW2, and XORW3.
-

-
- Miscellaneous instructions (9)—BICPSW, BISPSW, BPT, INDEX, MOVPSL, NOP, POPR, PUSH, and XFC.
-
- Procedure call instructions (3)—CALLG, CALLS, and RET.
-
- Queue instructions (6)—INSQHI, INSQTI, INSQUE, REMQHI, REMQTI, and REMQUE.
-
- Variable length bit field instructions (7)—CMPV, CMPZV, EXTV, EXTZV, FFC, FFS, and INSV.
-

▪ **Instruction Emulation**

Subset VAX processors and their operating systems cooperate to support emulation of those instructions that are omitted from the processor's instruction set. Programs running under the operating system can make use of these instructions as though they were supported directly by the processor. The process of emulating an omitted instruction depends on the instruction type. Emulation of string instructions is assisted by the processor, through the instruction emulation exception. Emulation of compatibility mode instructions and floating point instructions is done entirely by software.

▪ **MicroVAX I Systems**

The MicroVAX I is the first subset VAX system. There are two versions of the subset—one that includes F__ floating and G__ floating instructions, and one that includes F__ floating and D__ floating instructions. Neither version includes H__ floating instructions. The MicroVAX I processor includes some of the optional string instructions (CMPC3, LOCC, SCANC, SKPC, and SPANC), but does not include compatibility mode.

▪ **MicroVAX II Systems**

MicroVAX II is the first subset VAX system with the processor on a single chip. It includes the F__ floating, D__ floating, and G__ floating instructions in an optional floating-point unit (a separate chip), but does not include the H__ floating instructions. MicroVAX II includes none of the optional string instructions. It does not include optional processor registers or compatibility mode.

Chapter 11 • PDP-11 Compatibility Mode

During the design of the VAX computer architecture, Digital's engineers were acutely aware of the need to establish a high degree of compatibility with the large, well-established PDP-11 computer family. VAX systems represent the natural growth direction for many installations using PDP-11 machines and programs. It was important that VAX machines display selected compatibility features for good reasons—to ease the growth, to quicken program transition, and to protect customer investment. Also, VAX machines had to provide compatibility for people who would take advantage of its excellent program development tools. These tools are used to create and test programs that are to be run on PDP-11 systems. PDP-11 compatibility mode is now an option. VAX processors that implement compatibility mode do so as described in this chapter. Operating system software may emulate compatibility mode on processors that omit it. For details of the PDP-11 instruction set, see the *PDP-11 Architecture Handbook*.

NOTE

In this chapter, references to *compatibility mode* mean the PDP-11 compatibility mode of operation. References to *native mode* mean the VAX native mode of operation.

The PDP-11 compatibility mode makes a VAX computer look like a PDP-11 computer running PDP-11 instructions. Naturally, there are some restrictions and requirements. A VAX computer treats compatibility mode programs like other processes, and can run them in its multiprogramming environment along with native mode programs. The VAX computer should not be thought of as existing in one state or another, but rather as capable of handling both modes as needed.

If you are considering a VAX system for growth and for host program development, you will find that it provides useful compatibility with PDP-11 systems already in use or others that might be added. As a powerful link joining PDP-11 computers and VAX computers, compatibility mode can help you expand your computing resources efficiently. And programs that cannot take advantage of compatibility mode, for one reason or another, usually can be fixed easily and quickly.

What follows in this chapter is a fairly detailed review of the powers and the restrictions of the compatibility mode. Should you need a greater depth of information, your Digital Sales Representative or Software Specialist can supply it for you.

■ PDP-11 User Environment Emulation

Compatibility mode hardware, in conjunction with a compatibility mode software executive (which runs in native mode), can emulate the environment provided to user programs on a PDP-11 system. But this environment excludes from a complete PDP-11 the normal operation of the following features:

1. Privileged instructions such as HALT and RESET.
2. Special instructions such as traps and WAIT.
3. Access to internal processor registers (for example, processor status word and console switch register).
4. Direct access to trap and interrupt vectors.
5. Direct access to I/O devices. (PDP-11 compatibility mode programs can directly reference I/O devices if and only if proper mapping has been established by native mode software.)
6. Interrupt servicing.
7. Stack overflow protection.
8. Alternate general register sets.
9. The PDP-11 processor kernel and supervisor modes are not supported. The user mode is the only mode supported in PDP-11 compatibility mode.
10. Floating-point instructions.

Compatibility mode architecture is divided into two parts. The first part is the PDP-11 environment provided by the VAX hardware. Details of the operation of PDP-11 compatible operations can be found in the *PDP-11 Architecture Handbook*. The second part is the hardware provided in the VAX architecture that enable the implementation of various compatibility mode executives. This part is considered a subset of the VAX System Architecture.

General Registers

All the PDP-11 general registers and addressing modes are in the compatibility mode. Side effects caused by a destination address calculation have no effect on source values (except in JSR instructions), and autoincrement modes in JMP and JSR do not affect the new program counter. However, side effects caused by a source address calculation affect the value of a register used for destination address calculation. All PDP-11 addresses are 16 bits long. In compatibility mode, a 16-bit PDP-11 address is zero-extended to 32 bits.

The operands of some PDP-11 instructions are implied by the instruction type while others are specified as part of the instruction. Address mode operand specifiers include a 3-bit mode field specifying one of eight modes—autodecrement, autodecrement deferred, autoincrement, autoincrement deferred, index, index deferred, register, and register deferred.

Compatibility mode registers 0 through 6 are bits 15 through 0 of VAX general registers 0 through 6, respectively. Compatibility mode register 7 (program counter) is bit 15 through 0 of VAX general register 15. VAX registers 8 through 14 (stack pointer) are not affected by compatibility mode. When entering compatibility mode, VAX register 7 and the upper halves of registers 0 through 6 and 15 are ignored. When an exception or interrupt occurs from compatibility mode, VAX register 7 is *unpredictable* and the upper halves of R0 through R6 and the stacked R15 (PC) are zero. There are no floating accumulators (registers). That is why there are no FP11 floating point instructions in compatibility mode.

Stack Pointer Register

As in the PDP-11 processors, general register R6 is used as the stack pointer by certain instructions. However, it is not used by the hardware for any exceptions or interrupts, nor is there any stack overflow protection in compatibility mode.

Processor Status Word

A subset of the PDP-11 processor status word is available in compatibility mode. The format of the compatibility mode processor status word (PSW) is shown in Figure 11-1. Compatibility mode processor status word bits 0 through 4 have the same meaning as do the VAX processor status longword bits 0 through 4. They are the trace bit and the condition code bits.

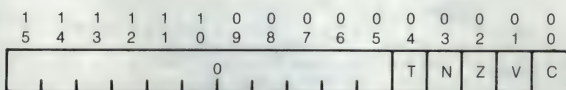


Figure 11-1 ■ Compatibility Mode Processor Status Word

The processor status word can be affected only by the RTI and RTT condition code instructions. When an RTI or RTT instruction is executed, bits 15 through 5 in the saved processor status word (PSW) on the stack are ignored.

Compatibility Mode Instructions

The compatibility mode instructions are listed in Table 11-1. Table 11-2 lists the trap instructions that cause the VAX processor to enter native mode. In native mode, either the complete trap may be serviced or the instruction may be simulated. Some instructions (such as WAIT and RESET) are considered reserved instructions in compatibility mode. When these instructions are encountered, they cause a fault to native mode. Table 11-3 lists the reserved instructions. In addition, all other opcodes not defined in Tables 11-1 and 11-2 result in a fault to native mode. No floating-point instructions are included in compatibility mode.

Table 11-1 • PDP-11 Compatibility Mode Instructions

Opcode (octal)	Mnemonic	Name
000002	RTI	Return from Interrupt
000006	RTT	Return from Trap
0001DD	JMP	Jump
00020R	RTS	Return from Subroutine
000240-000277	Condition Codes	
0003DD	SWAB	Swap Bytes
000400-003777	Branches	Branch
100000-103777	Branches	Branch
004RDD	JSR	Jump to Subroutine
X050DD	CLR(B)	Clear
X051DD	COM(B)	Complement
X052DD	INC(B)	Increment
X053DD	DEC(B)	Decrement
X054DD	NEG(B)	Negate
X055DD	ADC(B)	Add Carry
X056DD	SBC(B)	Subtract Carry
X057DD	TST(B)	Test
X060DD	ROR(B)	Rotate Right
X061DD	ROL(B)	Rotate Left
X062DD	ASR(B)	Arithmetic Shift Right
X063DD	ASL(B)	Arithmetic Shift Left
0065SS	MFPI*	Move from Previous Instruction Space

Table 11-1 • PDP-11 Compatibility Mode Instructions (Cont.)

Opcode (octal)	Mnemonic	Name
0066DD	MTP1*	Move to Previous Instruction Space
1065SS	MFPD*	Move from Previous Data Space
1066DD	MTPD*	Move to Previous Data Space
0067DD	SXT	Sign Extend Word
070RSS	MUL	Multiply
071RSS	DIV	Divide
072RSS	ASH	Arithmetic Shift
073RSS	ASHC	Arithmetic Shift Combined
074RSS	XOR	Exclusive OR
077RNN	SOB	Subtract One and Branch
X1SSDD	MOV(B)	Move
X2SSSS	CMP(B)	Compare
X3SSSS	BIT(B)	Bit Test
X4SSDD	BIC(B)	Bit Clear
X5SSDD	BIS(B)	Bit Set
06SSDD	ADD	Add
16SSDD	SUB	Subtract

Legend

DD Destination operand specifier

R Register specifier

SS Source operand specifier

X Operation specifier—0 for word, 1 for byte

* These instructions execute exactly as they would on a PDP-11 in user mode with Instruction and Data space overmapped. More specifically, they ignore the previous access level and act as if they were PUSH and POP instructions referencing the current stack.

Table 11-2 • PDP-11 Compatibility Mode Trap Instructions

Opcode (octal)	Mnemonic
000003	BPT
000004	IOT
104000-104377	EMT
104400-104777	TRAP

Table 11-3 • PDP-11 Compatibility Mode Reserved Instructions

Opcode (octal)	Mnemonic
000000	HALT
000001	WAIT
000005	RESET
000007	MFPT
00023N	SPL
0064NN	MARK
0070DD	CSM
07500R	FADD - FIS
07501R	FSUB - FIS
07502R	FMUL - FIS
07503R	FDIV - FIS
076XXX	Extended Instructions
1064SS	MTPS
1067DD	MFPS
17XXXX	FP11 Floating Point

▪ Entering and Leaving PDP-11 Compatibility Mode

Compatibility mode is entered when an REI instruction is executed with the compatibility mode bit of the processor status longword (PSL) on the stack is set. Other bits in the PSL either have the same effect as in native mode or are required to have specific values in compatibility mode. The effects of other bits in the PSL are listed in Table 11-4.

Table 11-4 • Effects of Processor Status Longword Bits

Bit	Effect
C	Condition Code
CUR MOD	Reserved operand fault if not 3
DV	Reserved operand fault if not zero
FPD	Reserved operand fault if not zero
FU	Reserved operand fault if not zero
IPL	Reserved operand fault if not zero
IS	Reserved operand fault if not zero
IV	Reserved operand fault if not zero
N	Condition Code
PRV MOD	Reserved operand fault if not 3
T	Trace bit
TP	Trace pending bit
V	Condition Code
Z	Condition Code

Native mode is reentered from compatibility mode on an exception or an interrupt. The processor status longword (PSL) pushed on the stack has all the bits that cause reserved operand faults set to the appropriate state.

Note that when an RTI or RTT instruction is executed in compatibility mode, the 11 high bits of the processor status word (PSW) are ignored. But when the PSW is restored as part of the PSL when going from native to compatibility mode, those bits must be zero or a reserved operand fault will occur.

▪ Memory Management

The PDP-11 uses 16-bit byte addresses. For this reason, compatibility mode programs are confined to execute in the first 64 Kbytes of the process part of virtual address space. There is a one-to-one correspondence between a compatibility mode virtual address and its VAX counterpart. For example, virtual address 0 references the same location in both modes. A compatibility mode address is interpreted as shown in Figure 11-2.

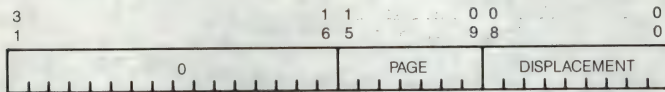


Figure 11-2 ■ Compatibility Mode Address Interpretation

The PDP-11 computers can provide different access protection to different segments of memory. PDP-11 segments are in 8-block increments. VAX segments are 512-byte pages. This is done because protection is specified by pages in the VAX architecture. (One VAX page equals eight PDP-11 blocks.)

The memory management system protects and relocates compatibility mode addresses in the normal manner. Thus, all of the memory management mechanisms available in native mode are available to the compatibility mode executive for managing both the virtual and physical memory of compatibility mode programs. All the exception conditions that can be caused by memory management in native mode can also occur when relocating a compatibility mode address.

Most of the features of the PDP-11 memory management hardware affecting the user environment can be simulated with the VAX memory management system. Table 11-5 provides a general description of how this can be done. Table 11-6 demonstrates how a PDP-11 environment can be created using the concepts in Table 11-5. There are 8 segments. Segments 0, 1, and 2 are program segments; 3 is unused; 4 and 5 are stack; 6 and 7 are read/write data.

Table 11-5 • PDP-11 Memory Management Simulation

PDP-11 Memory Management Feature	VAX Simulation Method
Eight segments per user	Eight segments can be simulated by dividing the 128 pages of the compatibility mode virtual address space into eight logical groups of 16 pages each having possibly different protection.
Segment size from 64 bytes to 8 Kbytes (1 to 128 blocks) in 64-byte increments, using contiguous memory	Segment size from 512 bytes to 8 Kbytes (1 to 16 pages) in 512-byte (1 page) increments, using discontinuous memory.
Forward growing segments (Expand Direction 0)	Can be simulated using page table entries specifying no access for those pages that are not allocated.
Backward growing segments (Expand Direction 1)	Can be simulated using page table entries specifying no access for those pages that are not allocated.
Segments begin on any 64-byte boundary	Segments begin on any 512-byte boundary.

Table 11-6 • Creating a PDP-11 Environment

PDP-11 Environment				VAX Page Table	
Segment Number	Size (bytes)	Expand Direction	Access Type	Page	Access Type
0	8K	Up	Read only	0-15	Read only
1	8K	Up	Read only	16-31	Read only
2	256	Up	Read only	32	Read only
3	0	None	None	33-77	No Access
4	1K	Down	Read/Write	78-79	Read/Write
5	8K	Down	Read/Write	80-95	Read/Write
6	8K	Up	Read/Write	96-111	Read/Write
7	2K	Up	Read/Write	112-115	Read/write
				116-127	No access

▪ Exceptions and Interrupts

All interrupts and exception conditions that occur while the machine is in compatibility mode cause the machine to enter native mode. Note that this includes backing up instruction side effects if necessary. The following exception conditions are specific to compatibility mode. All these exceptions create a three-longword frame on the kernel stack containing a processor status longword (PSL), program counter (PC), and one longword of trap-specific information. Bits 15 through 0 of this longword contain a code indicating the specific type of trap and bits 31 through 16 are zero. No compatibility mode exception conditions result in traps.

▪ Tracing in Compatibility Mode

A compatibility mode trace fault occurs at the beginning of an instruction when the trace bit is set in the processor status word at the beginning of the prior instruction. On trace faults, a 2-longword kernel stack frame is created. The frame contains the processor status longword and the program counter. The interrupt priority level (IPL) and interrupt stack (IS) bits are 0, and the compatibility mode (CM) bit is 1 in the stacked processor status longword. Compatibility mode trace faults use the same vector as native mode trace fault. In fact, the rules for trace fault generation in compatibility mode are identical to those for native mode. However, an odd address abort for an instruction fetch may precede the trace fault for that instruction. There are two ways to set the trace bit at the beginning of a compatibility mode instruction.

1. An RTT/RTI instruction is executed in compatibility mode and the trace bit in the processor status word image on the stack is set. In this case, the next instruction is executed and a trace fault is taken after that instruction. (The next instruction is the one pointed to by the program counter on the stack.)
2. An REI instruction is executed in native mode under the following conditions. The instruction has both the trace bit and the compatibility mode bit set and the trace pending bit clear in the saved processor status longword image on the stack. Again, one instruction is executed, and the trace trap is taken. (The operations that occur as a function of these conditions are the same whether or not compatibility mode is being entered from the REI instruction.)

▪ Unimplemented PDP-11 Traps

Some traps that occur in PDP-11 systems are not implemented in VAX systems PDP-11 compatibility mode.

1. There is no stack overflow trap. Stack overflow can be provided by the compatibility mode executive using the memory management mechanisms.
2. There is no concept of a double error trap in compatibility mode. This is because the first error always returns the machine to native mode.
3. All other trap conditions such as power failure, memory parity, and memory management traps cause the machine to enter native mode.

▪ Input/output References

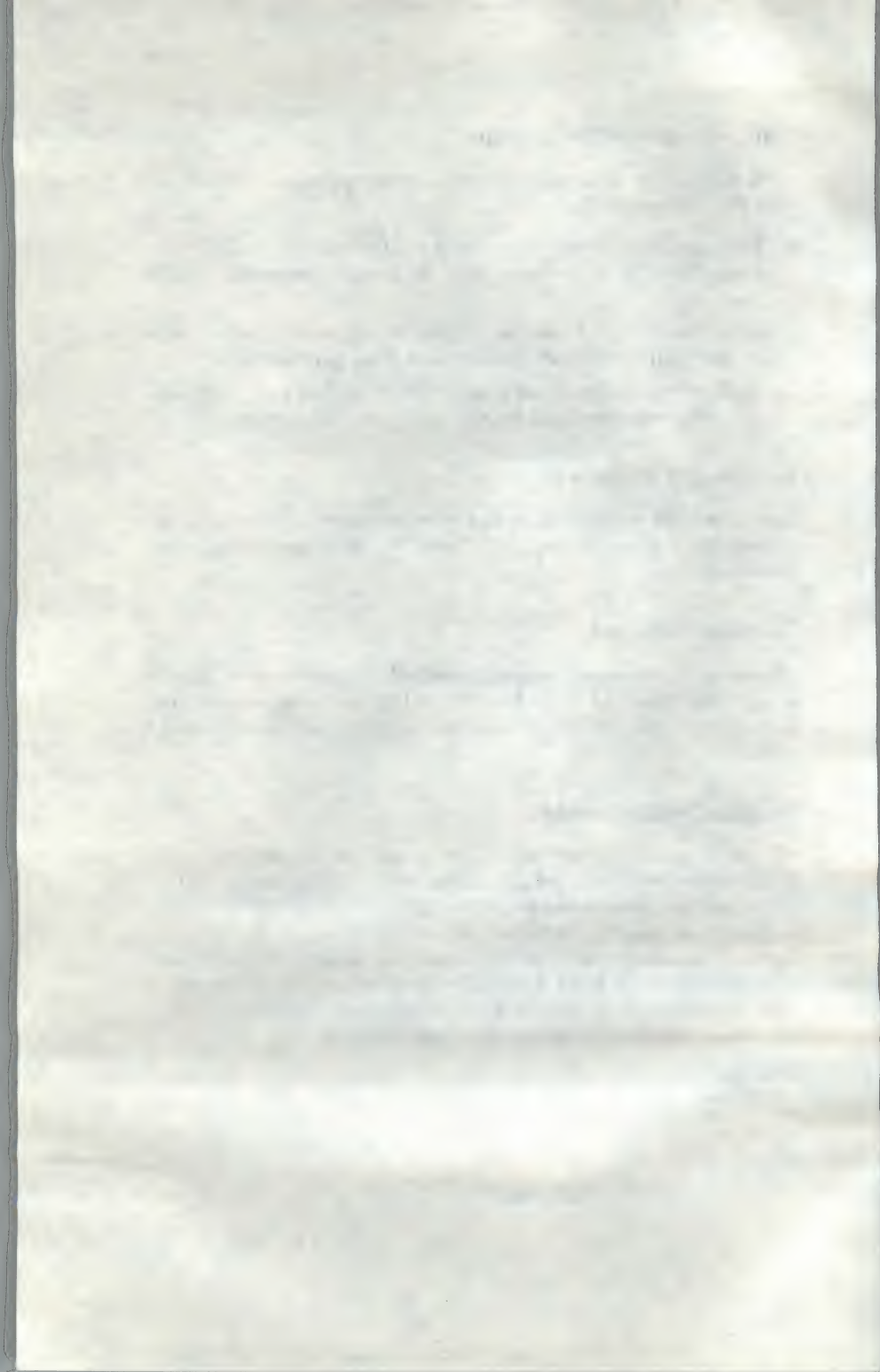
Instruction stream, data read, or data write references to I/O space are not allowed. The results are *unpredictable* if I/O space is referenced from compatibility mode.

▪ Processor Registers

The only processor register available in compatibility mode is part of the processor status word, and it may be referenced only with the condition code instructions, RTI and RTT. Access to all other registers must be done in native mode.

▪ Program Synchronization

All PDP-11 systems guarantee that read-modify-write operations to I/O device registers are interlocked. That is, the device can determine at the time of the read that the same register will be written as the next bus cycle. This synchronization also works in memory on most PDP-11 systems. In compatibility mode, instructions that have modify destinations perform this synchronization for UNIBUS I/O device registers but never for memory. Compatibility mode procedures can write data that is to be subsequently executed as an instruction without requiring additional synchronization.



Appendix A ■ Powers of Binary and Hexadecimal Numbers

Powers of Binary Numbers

Power	Number
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152
22	4,194,304
23	8,388,608
24	16,777,216
25	33,554,432
26	67,108,864
27	134,217,728

Powers of Binary Numbers (Cont.)

Power	Number
28	268,435,456
29	536,870,912
30	1,073,741,824
31	2,147,483,648
32	4,294,967,296
33	8,589,934,592
34	17,179,869,184
35	34,359,738,368

Powers of Hexadecimal Numbers

Power	Number
0	1
1	16
2	256
3	4,096
4	65,536
5	1,048,576
6	16,777,216
7	268,435,456
8	4,294,967,296
9	68,719,476,736
10	1,099,511,627,776
11	17,592,186,044,416
12	281,474,976,710,656
13	4,503,599,627,370,496
14	72,057,594,037,927,936
15	1,152,921,504,606,846,976

Appendix B ■ List of Instructions by Mnemonic

Mnemonic	Instruction	Opcode
ACBB	Add compare and branch byte	9D
ACBD	Add compare and branch D__ floating	6F
ACBF	Add compare and branch F__ floating	4F
ACBG	Add compare and branch G__ floating	4FFD
ACBH	Add compare and branch H__ floating	6FFD
ACBL	Add compare and branch longword	F1
ACBW	Add compare and branch word	3D
ADAWI	Add aligned word, interlocked	58
ADDB2	Add byte 2 operand	80
ADDB3	Add byte 3 operand	81
ADDD2	Add D__ floating 2 operand	60
ADDD3	Add D__ floating 3 operand	61
ADDF2	Add F__ floating 2 operand	40
ADDF3	Add F__ floating 3 operand	41
ADDG2	Add G__ floating 2 operand	40FD
ADDG3	Add G__ floating 3 operand	41FD
ADDH2	Add H__ floating 2 operand	60FD
ADDH3	Add H__ floating 3 operand	61FD
ADDL2	Add longword 2 operand	C0
ADDL3	Add longword 3 operand	C1
ADDP4	Add packed 4 operand	20
ADDP6	Add packed 6 operand	21
ADDW2	Add word 2 operand	A0
ADDW3	Add word 3 operand	A1
ADWC	Add with carry	D8
AOBLEQ	Add one and branch on less or equal	F3
AOBLSS	Add one and branch on less	F2
ASHL	Arithmetic shift longword	78

Mnemonic	Instruction	Opcode
ASHP	Arithmetic shift and round packed	F8
ASHQ	Arithmetic shift quadword	79
BBC	Branch on bit clear	E1
BBCC	Branch on bit clear and clear	E5
BBCCI	Branch on bit clear and clear, interlocked	E7
BBCS	Branch on bit clear and set	E3
BBS	Branch on bit set	E0
BBSC	Branch on bit set and clear	E4
BBSS	Branch on bit set and set	E2
BBSSI	Branch on bit set and set, interlocked	E6
BCC	Branch on carry clear	1E
BCS	Branch on carry set	1F
BEQL	Branch on equal	13
BEQLU	Branch on equal, unsigned	13
BGEQ	Branch on greater or equal	18
BGEQU	Branch on greater or equal, unsigned	1E
BGTR	Branch on greater	14
BGTRU	Branch on greater, unsigned	1A
BICB2	Bit clear byte 2 operand	8A
BICB3	Bit clear byte 3 operand	8B
BICL2	Bit clear longword 2 operand	CA
BICL3	Bit clear longword 3 operand	CB
BICPSW	Bit clear processor status word	B9
BICW2	Bit clear word 2 operand	AA
BICW3	Bit clear word 3 operand	AB
BISB2	Bit set byte 2 operand	88
BISB3	Bit set byte 3 operand	89
BISL2	Bit set longword 2 operand	C8
BISL3	Bit set longword 3 operand	C9
BISPSW	Bit set processor status word	B8
BISW2	Bit set word 2 operand	A8
BISW3	Bit set word 3 operand	A9
BITB	Bit test byte	93

Mnemonic	Instruction	Opcode
BITL	Bit test longword	D3
BITW	Bit test word	B3
BLBC	Branch on low bit clear	E9
BLBS	Branch on low bit set	E8
BLEQ	Branch on less or equal	15
BLEQU	Branch on less or equal, unsigned	1B
BLSS	Branch on less	19
BLSSU	Branch on less, unsigned	1F
BNEQ	Branch on not equal	12
BNEQU	Branch on not equal, unsigned	12
BPT	Breakpoint fault	03
BRB	Branch with byte displacement	11
BRW	Branch with word displacement	31
BSBB	Branch to subroutine with byte displacement	10
BSBW	Branch to subroutine with word displacement	30
BUGL	Bugcheck longword	FDFE
BUGW	Bugcheck word	FEFF
BVC	Branch on overflow clear	1C
BVS	Branch on overflow set	1D
CALLG	Call with general argument list	FA
CALLS	Call with stack	FB
CASE	BCase byte	8F
CASEL	Case longword	CF
CASEW	Case word	AF
CHME	Change mode to executive	BD
CHMK	Change mode to kernel	BC
CHMS	Change mode to supervisor	BE
CHMU	Change mode to user	BF
CLRB	Clear byte	94
CLRD	Clear D__ floating	7C
CLRF	Clear F__ floating	D4
CLRG	Clear G__ floating	7C
CLRHI	Clear H__ floating	7CFD

Mnemonic	Instruction	Opcode
CLRL	Clear longword	D4
CLRO	Clear octaword	7CFD
CLRQ	Clear quadword	7C
CLRW	Clear word	B4
CMPB	Compare byte	91
CMPC3	Compare character 3 operand	29
CMPC5	Compare character 5 operand	2D
CPMD	Compare D__ floating	71
CMPF	Compare F__ floating	51
CMPG	Compare G__ floating	51FD
CMPH	Compare H__ floating	71FD
CMPL	Compare longword	D1
CMPP3	Compare packed 3 operand	35
CMPP4	Compare packed 4 operand	37
CMPV	Compare field	EC
CMPW	Compare word	B1
CMPZV	Compare zero-extended field	ED
CRC	Calculate cyclic redundancy check	0B
CVTBD	Convert byte to D__ floating	6C
CVTBF	Convert byte to F__ floating	4C
CVTBG	Convert byte to G__ floating	4CFD
CVTBH	Convert byte to H__ floating	6CFD
CVTBL	Convert byte to longword	98
CVTBW	Convert byte to word	99
CVTDB	Convert D__ floating to byte	68
CVTDF	Convert D__ floating to F__ floating	76
CVTDH	Convert D__ floating to H__ floating	32FD
CVTDL	Convert D__ floating to longword	6A
CVTDW	Convert D__ floating to word	69
CVTFB	Convert F__ floating to byte	48
CVTFD	Convert F__ floating to D__ floating	56
CVTFG	Convert F__ floating to G__ floating	99FD
CVTFH	Convert F__ floating to H__ floating	98FD

Mnemonic	Instruction	Opcode
CVTFL	Convert F__ floating to longword	4A
CVTFW	Convert F__ floating to word	49
CVTGB	Convert G__ floating to byte	48FD
CVTGF	Convert G__ floating to F__ floating	33FD
CVTGH	Convert G__ floating to H__ floating	56FD
CVTGL	Convert G__ floating to longword	4AFD
CVTGW	Convert G__ floating to word	49FD
CVTHB	Convert H__ floating to byte	68FD
CVTHD	Convert H__ floating to D__ floating	F7FD
CVTHF	Convert H__ floating to F__ floating	F6FD
CVTHG	Convert H__ floating to G__ floating	76FD
CVTHL	Convert H__ floating to longword	6AFD
CVTHW	Convert H__ floating to word	69FD
CVTLB	Convert longword to byte	F6
CVTLD	Convert longword to D__ floating	6E
CVTLF	Convert longword to F__ floating	4E
CVTLG	Convert longword to G__ floating	4EFD
CVTLH	Convert longword to H__ floating	6EFD
CVTLP	Convert longword to packed	F9
CVTLW	Convert longword to word	F7
CVTPL	Convert packed to longword	36
CVTPT	Convert packed to trailing numeric	24
CVTPS	Convert packed to leading separate numeric	08
CVTRDL	Convert rounded D__ floating to longword	6B
CVTRFL	Convert rounded F__ floating to longword	4B
CVTRGL	Convert rounded G__ floating to longword	4BFD
CVTRHL	Convert rounded H__ floating to longword	6BFD
CVTSP	Convert leading separate numeric to packed	09
CVTTP	Convert trailing numeric to packed	26
CVTWB	Convert word to byte	33
CVTWD	Convert word to D__ floating	6D
CVTWF	Convert word to F__ floating	4D
CVTWG	Convert word to G__ floating	4DFD

Mnemonic	Instruction	Opcode
CVTWH	Convert word to H__ floating	6DFD
CVTWL	Convert word to longword	32
DECB	Decrement byte	97
DECL	Decrement longword	D7
DECW	Decrement word	B7
DIVB2	Divide byte 2 operand	86
DIVB3	Divide byte 3 operand	87
DIVD2	Divide D__ floating 2 operand	66
DIVD3	Divide D__ floating 3 operand	67
DIVF2	Divide F__ floating 2 operand	46
DIVF3	Divide F__ floating 3 operand	47
DIVG2	Divide G__ floating 2 operand	46FD
DIVG3	Divide G__ floating 3 operand	47FD
DIVH2	Divide H__ floating 2 operand	66FD
DIVH3	Divide H__ floating 3 operand	67FD
DIVL2	Divide longword 2 operand	C6
DIVL3	Divide longword 3 operand	C7
DIVP	Divide packed	27
DIVW2	Divide word 2 operand	A6
DIVW3	Divide word 3 operand	A7
EDITPC	Edit packed to character	38
EDIV	Extended divide	7B
EMODD	Extended modulus D__ floating	74
EMODF	Extended modulus F__ floating	54
EMODG	Extended modulus G__ floating	54FD
EMODH	Extended modulus H__ floating	74FD
EMUL	Extended multiply	7A
EXTV	Extract field	EE
EXTZV	Extract zero-extended field	EF
FFC	Find first clear bit	EB
FFS	Find first set bit	EA
HALT	Halt	00
INCB	Increment byte	96

Mnemonic	Instruction	Opcode
INCL	Increment longword	D6
INCW	Increment word	B6
INDEX	Compute index	0A
INSQHI	Insert into queue head, interlocked	5C
INSQTI	Insert into queue tail, interlocked	5D
INSQUE	Insert into queue	0E
INSV	Insert field	F0
JMP	Jump	17
JSB	Jump to subroutine	16
LDPCTX	Load process context	06
LOCC	Locate character	3A
MATCHC	Match characters	39
MCOMB	Move complemented byte	92
MCOML	Move complemented longword	D2
MCOMW	Move complemented word	B2
MFPR	Move from processor register	DB
MNEGB	Move negated byte	8E
MNEGD	Move negated D__ floating	72
MNEGF	Move negated F__ floating	52
MNEGG	Move negated G__ floating	52FD
MNEGH	Move negated H__ floating	72FD
MNEGL	Move negated longword	CE
MNEGW	Move negated word	AE
MOVAB	Move address of byte	9E
MOVAD	Move address of D__ floating	7E
MOVAF	Move address of F__ floating	DE
MOVAG	Move address of G__ floating	7E
MOVAH	Move address of H__ floating	7EFD
MOVAL	Move address of longword	DE
MOVAO	Move address of octaword	7EFD
MOVAQ	Move address of quadword	7E
MOVAW	Move address of word	3E
MOVB	Move byte	90

Mnemonic	Instruction	Opcode
MOVC3	Move character 3 operand	28
MOVC5	Move character 5 operand	2C
MOVD	Move D__ floating	70
MOVF	Move F__ floating	50
MOVG	Move G__ floating	50FD
MOVH	Move H__ floating	70FD
MOVL	Move longword	D0
MOVO	Move octaword	7DFD
MOVP	Move packed	34
MOVPSL	Move processor status longword	DC
MOVQ	Move quadword	7D
MOVTC	Move translated characters	2E
MOVTUC	Move translated until character	2F
MOVW	Move word	B0
MOVZBL	Move zero-extended byte to longword	9A
MOVZBW	Move zero-extended byte to word	9B
MOVZWL	Move zero-extended word to longword	3C
MTPR	Move to processor register	DA
MULB2	Multiply byte 2 operand	84
MULB3	Multiply byte 3 operand	85
MULD2	Multiply D__ floating 2 operand	64
MULD3	Multiply D__ floating 3 operand	65
MULF2	Multiply F__ floating 2 operand	44
MULF3	Multiply F__ floating 3 operand	45
MULG2	Multiply G__ floating 2 operand	44FD
MULG3	Multiply G__ floating 3 operand	45FD
MULH2	Multiply H__ floating 2 operand	64FD
MULH3	Multiply H__ floating 3 operand	65FD
MULL2	Multiply longword 2 operand	C4
MULL3	Multiply longword 3 operand	C5
MULP	Multiply packed	25
MULW2	Multiply word 2 operand	A4
MULW3	Multiply word 3 operand	A5

Mnemonic	Instruction	Opcode
NOP	No operation	01
POLYD	Polynomial evaluate D__ floating	75
POLYF	Polynomial evaluate F__ floating	55
POLYG	Polynomial evaluate G__ floating	55FD
POLYH	Polynomial evaluate H__ floating	75FD
POPR	Pop registers	BA
PROBER	Probe read access	0C
PROBEW	Probe write access	0D
PUSHAB	Push address byte	9F
PUSHAD	Push address of D__ floating	7F
PUSHAF	Push address of F__ floating	DF
PUSHAG	Push address of G__ floating	7F
PUSHAH	Push address of H__ floating	7FFD
PUSHAL	Push address of longword	DF
PUSHAO	Push address of octaword	7FFD
PUSHAQ	Push address of quadword	7F
PUSHAW	Push address of word	3F
PUSHL	Push longword	DD
PUSHR	Push registers	BB
REI	Return from exception or interrupt	02
REMQHI	Remove from queue head, interlocked	5E
REMQTI	Remove from queue tail, interlocked	5F
REMQUE	Remove from queue	0F
RET	Return from called procedure	04
ROTL	Rotate longword	9C
RSB	Return from subroutine	05
SBWC	Subtract with carry	D9
SCANC	Scan for character	2A
SKPC	Skip character	3B
SOBGEQ	Subtract one and branch on greater or equal	F4
SOBGTR	Subtract one and branch on greater	F5
SPANC	Span characters	2B
SUBB2	Subtract byte 2 operand	82

Mnemonic	Instruction	Opcode
SUBB3	Subtract byte 3 operand	83
SUBD2	Subtract D__ floating 2 operand	62
SUBD3	Subtract D__ floating 3 operand	63
SUBF2	Subtract F__ floating 2 operand	42
SUBF3	Subtract F__ floating 3 operand	43
SUBG2	Subtract G__ floating 2 operand	42FD
SUBG3	Subtract G__ floating 3 operand	43FD
SUBH2	Subtract H__ floating 2 operand	62FD
SUBH3	Subtract H__ floating 3 operand	63FD
SUBL2	Subtract longword 2 operand	C2
SUBL3	Subtract longword 3 operand	C3
SUBP4	Subtract packed 4 operand	22
SUBP6	Subtract packed 6 operand	23
SUBW2	Subtract word 2 operand	A2
SUBW3	Subtract word 3 operand	A3
SVPCTX	Save process context	07
TSTB	Test byte	95
TSTD	Test D__ floating	73
TSTF	Test F__ floating	53
TSTG	Test G__ floating	53FD
TSTH	Test H__ floating	73FD
TSTL	Test longword	D5
TSTW	Test word	B5
XFC	Extended function call	FC
XORB2	Exclusive OR byte 2 operand	8C
XORB3	Exclusive OR byte 3 operand	8D
XORL2	Exclusive OR longword 2 operand	CC
XORL3	Exclusive OR longword 3 operand	CD
XORW2	Exclusive OR word 2 operand	AC
XORW3	Exclusive OR word 3 operand	AD

Appendix C ■ List of Instructions by Opcode

Opcode	Mnemonic	Instruction
00	HALT	Halt
01	NOP	No operation
02	REI	Return from exception or interrupt
03	BPT	Breakpoint fault
04	RET	Return from called procedure
05	RSB	Return from subroutine
06	LDPCTX	Load process context
07	SVPCTX	Save process context
08	CVTPS	Convert packed to leading separate numeric
09	CVTSP	Convert leading separate numeric to packed
0A	INDEX	Compute index
0B	CRC	Calculate cyclic redundancy check
0C	PROBER	Probe read access
0D	PROBEW	Probe write access
0E	INSQUE	Insert into queue
0F	REMQUE	Remove from queue
10	BSBB	Branch to subroutine with byte displacement
11	BRB	Branch with byte displacement
12	BNEQ	Branch on not equal
12	BNEQU	Branch on not equal, unsigned
13	BEQL	Branch on equal
13	BEQLU	Branch on equal, unsigned
14	BGTR	Branch on greater
15	BLEQ	Branch on less or equal
16	JSB	Jump to subroutine
17	JMP	Jump
18	BGEQ	Branch on greater or equal
19	BLSS	Branch on less

Opcode	Mnemonic	Instruction
1A	BGTRU	Branch on greater, unsigned
1B	BLEQU	Branch on less or equal, unsigned
1C	BVC	Branch on overflow clear
1D	BVS	Branch on overflow set
1E	BGEQU	Branch on greater or equal, unsigned
1E	BCC	Branch on carry clear
1F	BLSSU	Branch on less, unsigned
1F	BCS	Branch on carry set
20	ADDP4	Add packed 4 operand
21	ADDP6	Add packed 6 operand
22	SUBP4	Subtract packed 4 operand
23	SUBP6	Subtract packed 6 operand
24	CVTPT	Convert packed to trailing numeric
25	MULP	Multiply packed
26	CVTTP	Convert trailing numeric to packed
27	DIVP	Divide packed
28	MOVC3	Move character 3 operand
29	CMPC3	Compare character 3 operand
2A	SCANC	Scan for character
2B	SPANC	Span characters
2C	MOVC5	Move character 5 operand
2D	CMPC5	Compare character 5 operand
2E	MOVTC	Move translated characters
2F	MOVTUC	Move translated until character
30	BSBW	Branch to subroutine with word displacement
31	BRW	Branch with word displacement
32	CVTWL	Convert word to longword
33	CVTWB	Convert word to byte
34	MOVP	Move packed
35	CMPP3	Compare packed 3 operand
36	CVTPL	Convert packed to longword
37	CMPP4	Compare packed 4 operand
38	EDITPC	Edit packed to character

Opcode	Mnemonic	Instruction
39	MATCHC	Match characters
3A	LOCC	Locate character
3B	SKPC	Skip character
3C	MOVZWL	Move zero-extended word to longword
3D	ACBW	Add compare and branch word
3E	MOVAW	Move address of word
3F	PUSHAW	Push address of word
40	ADDF2	Add F__ floating 2 operand
41	ADDF3	Add F__ floating 3 operand
42	SUBF2	Subtract F__ floating 2 operand
43	SUBF3	Subtract F__ floating 3 operand
44	MULF2	Multiply F__ floating 2 operand
45	MULF3	Multiply F__ floating 3 operand
46	DIVF2	Divide F__ floating 2 operand
47	DIVF3	Divide F__ floating 3 operand
48	CVTFB	Convert F__ floating to byte
49	CVTFW	Convert F__ floating to word
4A	CVTFL	Convert F__ floating to longword
4B	CVTRFL	Convert rounded F__ floating to longword
4C	CVTBF	Convert byte to F__ floating
4D	CVTWF	Convert word to F__ floating
4E	CVTLF	Convert longword to F__ floating
4F	ACBF	Add compare and branch floating
50	MOVF	Move F__ floating
51	CMPF	Compare F__ floating
52	MNEGF	Move negated F__ floating
53	TSTF	Test F__ floating
54	EMODF	Extended modulus F__ floating
55	POLYF	Polynomial evaluate F__ floating
56	CVTFD	Convert F__ floating to D__ floating
57	<i>Reserved</i>	
58	ADAWI	Add aligned word, interlocked
59	<i>Reserved</i>	

Opcode	Mnemonic	Instruction
5A	<i>Reserved</i>	
5B	<i>Reserved</i>	
5C	INSQHI	Insert into queue head, interlocked
5D	INSQTI	Insert into queue tail, interlocked
5E	REMQHI	Remove from queue head, interlocked
5F	REMQTI	Remove from queue tail, interlocked
60	ADDD2	Add D__ floating 2 operand
61	ADDD3	Add D__ floating 3 operand
62	SUBD2	Subtract D__ floating 2 operand
63	SUBD3	Subtract D__ floating 3 operand
64	MULD2	Multiply D__ floating 2 operand
65	MULD3	Multiply D__ floating 3 operand
66	DIVD2	Divide D__ floating 2 operand
67	DIVD3	Divide D__ floating 3 operand
68	CVTDB	Convert D__ floating to byte
69	CVTDW	Convert D__ floating to word
6A	CVTDL	Convert D__ floating to longword
6B	CVTRDL	Convert rounded D__ floating to longword
6C	CVTBD	Convert byte to D__ floating
6D	CVTWD	Convert word to D__ floating
6E	CVTLD	Convert longword to D__ floating
6F	ACBD	Add compare and branch D__ floating
70	MOVD	Move D__ floating
71	CMPD	Compare D__ floating
72	MNEGD	Move negated D__ floating
73	TSTD	Test D__ floating
74	EMODD	Extended modulus D__ floating
75	POLYD	Polynomial evaluate D__ floating
76	CVTDF	Convert D__ floating to F__ floating
77	<i>Reserved</i>	
78	ASHL	Arithmetic shift longword
79	ASHQ	Arithmetic shift quadword
7A	EMUL	Extended multiply

Opcode	Mnemonic	Instruction
7B	EDIV	Extended divide
7C	CLRQ	Clear quadword
7C	CLRD	Clear D__ floating
7C	CLRG	Clear G__ floating
7D	MOVQ	Move quadword
7E	MOVAQ	Move address of quadword
7E	MOVAD	Move address of D__ floating
7E	MOVAG	Move address of G__ floating
7F	PUSHAQ	Push address of quadword
7F	PUSHAD	Push address of D__ floating
7F	PUSHAG	Push address of G__ floating
80	ADDB2	Add byte 2 operand
81	ADDB3	Add byte 3 operand
82	SUBB2	Subtract byte 2 operand
83	SUBB3	Subtract byte 3 operand
84	MULB2	Multiply byte 2 operand
85	MULB3	Multiply byte 3 operand
86	DIVB2	Divide byte 2 operand
87	DIVB3	Divide byte 3 operand
88	BISB2	Bit set byte 2 operand
89	BISB3	Bit set byte 3 operand
8A	BICB2	Bit clear byte 2 operand
8B	BICB3	Bit clear byte 3 operand
8C	XORB2	Exclusive OR byte 2 operand
8D	XORB3	Exclusive OR byte 3 operand
8E	MNEGB	Move negated byte
8F	CASEB	Case byte
90	MOVB	Move byte
91	CMPB	Compare byte
92	MCOMB	Move complemented byte
93	BITB	Bit test byte
94	CLRB	Clear byte
95	TSTB	Test byte

Opcode	Mnemonic	Instruction
96	INCB	Increment byte
97	DECB	Decrement byte
98	CVTBL	Convert byte to longword
99	CVTBW	Convert byte to word
9A	MOVZBL	Move zero-extended byte to longword
9B	MOVZBW	Move zero-extended byte to word
9C	ROTL	Rotate longword
9D	ACBB	Add compare and branch byte
9E	MOVAB	Move address of byte
9F	PUSHAB	Push address of byte
A0	ADDW2	Add word 2 operand
A1	ADDW3	Add word 3 operand
A2	SUBW2	Subtract word 2 operand
A3	SUBW3	Subtract word 3 operand
A4	MULW2	Multiply word 2 operand
A5	MULW3	Multiply word 3 operand
A6	DIVW2	Divide word 2 operand
A7	DIVW3	Divide word 3 operand
A8	BISW2	Bit set word 2 operand
A9	BISW3	Bit set word 3 operand
AA	BICW2	Bit clear word 2 operand
AB	BICW3	Bit clear word 3 operand
AC	XORW2	Exclusive OR word 2 operand
AD	XORW3	Exclusive OR word 3 operand
AE	MNEGW	Move negated word
AF	CASEW	Case word
B0	MOVW	Move word
B1	CMPW	Compare word
B2	MCOMW	Move complemented word
B3	BITW	Bit test word
B4	CLRW	Clear word
B5	TSTW	Test word
B6	INCW	Increment word

Opcode	Mnemonic	Instruction
B7	DECW	Decrement word
B8	BISPSW	Bit set processor status word
B9	BICPSW	Bit clear processor status word
BA	POPR	Pop register
BB	PUSHR	Push register
BC	CHMK	Change mode to kernel
BD	CHME	Change mode to executive
BE	CHMS	Change mode to supervisor
BF	CHMU	Change mode to user
C0	ADDL2	Add longword 2 operand
C1	ADDL3	Add longword 3 operand
C2	SUBL2	Subtract longword 2 operand
C3	SUBL3	Subtract longword 3 operand
C4	MULL2	Multiply longword 2 operand
C5	MULL3	Multiply longword 3 operand
C6	DIVL2	Divide longword 2 operand
C7	DIVL3	Divide longword 3 operand
C8	BISL2	Bit set longword 2 operand
C9	BISL3	Bit set longword 3 operand
CA	BICL2	Bit clear longword 2 operand
CB	BICL3	Bit clear longword 3 operand
CC	XORL2	Exclusive OR longword 2 operand
CD	XORL3	Exclusive OR longword 3 operand
CE	MNEGL	Move negated longword
CF	CASEL	Case longword
D0	MOVL	Move longword
D1	CMPL	Compare longword
D2	MCOML	Move complemented longword
D3	BITL	Bit test longword
D4	CLRL	Clear longword
D4	CLRF	Clear F__ floating
D5	TSTL	Test longword
D6	INCL	Increment longword

Opcode	Mnemonic	Instruction
D7	DECL	Decrement longword
D8	ADWC	Add with carry
D9	SBWC	Subtract with carry
DA	MTPR	Move to processor register
DB	MFPR	Move from processor register
DC	MOVPSL	Move processor status longword
DD	PUSHL	Push longword
DE	MOVAL	Move address of longword
DE	MOVAF	Move address of F__ floating
DF	PUSHAL	Push address of longword
DF	PUSHAF	Push address of F__ floating
E0	BBS	Branch on bit set
E1	BBC	Branch on bit clear
E2	BBSS	Branch on bit set and set
E3	BBCS	Branch on bit clear and set
E4	BBSC	Branch on bit set and clear
E5	BBCC	Branch on bit clear and clear
E6	BBSSI	Branch on bit set and set, interlocked
E7	BBCCI	Branch on bit clear and clear, interlocked
E8	BLBS	Branch on low bit set
E9	BLBC	Branch on low bit clear
EA	FFS	Find first set bit
EB	FFC	Find first clear bit
EC	CMPV	Compare field
ED	CMPZV	Compare zero-extended field
EE	EXTV	Extract field
EF	EXTZV	Extract zero-extended field
F0	INSV	Insert field
F1	ACBL	Add compare and branch longword
F2	AOBLSS	Add one and branch on less
F3	AOBLEQ	Add one and branch on less or equal
F4	SOBGEQ	Subtract one and branch on greater or equal
F5	SOBGTR	Subtract one and branch on greater

Opcode	Mnemonic	Instruction
F6	CVTLB	Convert longword to byte
F7	CVTLW	Convert longword to word
F8	ASHP	Arithmetic shift and round packed
F9	CVTLP	Convert longword to packed
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack argument list
FC	XFC	Extended function call
FD	<i>Reserved</i>	Escape to 2-byte opcode
FE	<i>Reserved</i>	Escape to 2-byte opcode
FF	<i>Reserved</i>	Escape to 2-byte opcode
00FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
31FD	<i>Reserved</i>	
32FD	CVTDH	Convert D__ floating to H__ floating
33FD	CVTGF	Convert G__ floating to F__ floating
34FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
3FFD	<i>Reserved</i>	
40FD	ADDG2	Add G__ floating 2 operand
41FD	ADDG3	Add G__ floating 3 operand
42FD	SUBG2	Subtract G__ floating 2 operand
43FD	SUBG3	Subtract G__ floating 3 operand
44FD	MULG2	Multiply G__ floating 2 operand
45FD	MULG3	Multiply G__ floating 3 operand
46FD	DIVG2	Divide G__ floating 2 operand
47FD	DIVG3	Divide G__ floating 3 operand
48FD	CVTGB	Convert G__ floating to byte
49FD	CVTGW	Convert G__ floating to word
4AFD	CVTGL	Convert G__ floating to longword

Opcode	Mnemonic	Instruction
4BFD	CVTRGL	Convert rounded G__ floating to longword
4CFD	CVTBG	Convert byte to G__ floating
4DFD	CVTWG	Convert word to G__ floating
4EFD	CVTLG	Convert longword to G__ floating
4FFD	ACBG	Add compare and branch G__ floating
50FD	MOVG	Move G__ floating
51FD	CMPG	Compare G__ floating
52FD	MNEGG	Move negated G__ floating
53FD	TSTG	Test G__ floating
54FD	EMODG	Extended modulus G__ floating
55FD	POLYG	Polynomial evaluate G__ floating
56FD	CVTGH	Convert G__ floating to H__ floating
57FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
5FFD	<i>Reserved</i>	
60FD	ADDH2	Add H__ floating 2 operand
61FD	ADDH3	Add H__ floating 3 operand
62FD	SUBH2	Subtract H__ floating 2 operand
63FD	SUBH3	Subtract H__ floating 3 operand
64FD	MULH2	Multiply H__ floating 2 operand
65FD	MULH3	Multiply H__ floating 3 operand
66FD	DIVH2	Divide H__ floating 2 operand
67FD	DIVH3	Divide H__ floating 3 operand
68FD	CVTHB	Convert H__ floating to byte
69FD	CVTHW	Convert H__ floating to word
6AFD	CVTHL	Convert H__ floating to longword
6BFD	CVTRHL	Convert rounded H__ floating to longword
6CFD	CVTBH	Convert byte to H__ floating
6DFD	CVTWH	Convert word to H__ floating
6EFD	CVTLH	Convert longword to H__ floating
6FFD	ACBH	Add compare and branch H__ floating

Opcode	Mnemonic	Instruction
70FD	MOVH	Move H__ floating
71FD	CMPH	Compare H__ floating
72FD	MNEGH	Move negated H__ floating
73FD	TSTH	Test H__ floating
74FD	EMODH	Extended modulus H__ floating
75FD	POLYH	Polynomial evaluate H__ floating
76FD	CVTHG	Convert H__ floating to G__ floating
77FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
7BFD	<i>Reserved</i>	
7CFD	CLRH	Clear H__ floating
7CFD	CLRO	Clear octaword
7DFD	MOV0	Move octaword
7EFD	MOVAH	Move address of H__ floating
7EFD	MOVAO	Move address of octaword
7FFD	PUSHAH	Push address of H__ floating
7FFD	PUSHAO	Push address of octaword
80FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
8FFD	<i>Reserved</i>	
90FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
97FD	<i>Reserved</i>	
98FD	CVTFH	Convert F__ floating to H__ floating
99FD	CVTFG	Convert F__ floating to G__ floating

Opcode	Mnemonic	Instruction
9AFD	<i>Reserved</i>	
.	.	
.	.	
.	.	
9FFD	<i>Reserved</i>	
A0FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
EFFD	<i>Reserved</i>	
F0FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
F5FD	<i>Reserved</i>	
F6FD	CVTHF	Convert H__ floating to F__ floating
F7FD	CVTHD	Convert H__ floating to D__ floating
F8FD	<i>Reserved</i>	
.	.	
.	.	
.	.	
FFFD	<i>Reserved</i>	
00FF	<i>Reserved</i>	
.	.	
.	.	
.	.	
FCFF	<i>Reserved</i>	
FDFE	BUGL	Bugcheck longword
FEFF	BUGW	Bugcheck word
FFFF	<i>Reserved</i>	

Glossary

abort: An exception that occurs in the middle of an instruction that can leave the registers and memory in an indeterminate state. When in this state, the instruction may not be able to be restarted.

absolute indexed mode: An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode: Autoincrement deferred mode in which the program counter (PC) is used as the register. The PC contains the address of the location containing the actual operand.

access mode: Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected, kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3).

access type: (1) How the processor accesses instruction operands. Access types are read, write, modify, address, and branch. (2) The way in which a procedure accesses its arguments.

access violation: (1) An attempt to reference an address that is not mapped into virtual memory. (2) An attempt to reference an address that is not accessible by the current access mode.

address: A number used by the operating system and user software to identify a storage location. *See also* **virtual address**, **physical address**.

address access type: A type of operation in which the specified operand of an instruction is not directly accessed when the processor executes the instruction. The context of the address calculation is given by the data type of the operand.

addressing mode: The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers.

address space: The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses.

alphanumeric character: An uppercase or lowercase letter (A to Z, a to z), a dollar sign (\$), an underscore (_), or a decimal digit (0 to 9).

American Standard Code for Information Interchange (ASCII): A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, control, and other special symbols used in text representation and communications protocol.

Argument Pointer: General register 12 (R12). By convention, the argument pointer (AP) contains the address of the base of the argument list for procedures initiated using the CALL instructions.

autodecrement index mode: A mode in which the base operand specifier uses autodecrement mode addressing.

autodecrement mode: A mode in which the contents of the selected register are decremented, and the result is used as the address of the actual operand of the instruction. The contents of the register are decremented according to the data type context of the register—1 for byte; 2 for word; 4 for longword and F__ floating; 8 for quadword, G__ floating, and D__ floating; and 16 for octaword and H__ floating.

auto deferred indexed mode: An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

autoincrement deferred mode: An addressing mode in which the specified register contains the address of a longword that contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the program counter is used as the register, this mode is called absolute mode.

autoincrement indexed mode: An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

autoincrement mode: A mode in which the contents of the specified register are used as the address of the operand; then the contents of the register are incremented by the size of the operand.

balance set: The set of all process working sets currently resident in physical memory. The processes whose working sets are in the balance set have memory requirements that balance with available memory.

base operand address: The address of the base of a table or array referenced by index mode addressing.

base operand specifier: The register used to calculate the base operand address of a table or array referenced by index mode addressing.

base register: A general register used to contain the address of the first entry in a list, table, array, or other data structure.

bit complement: The result of exchanging 0s and 1s in the binary representation of a number. Thus the bit complement of the binary number 11011001 (217 (decimal)) is 00100110. Bit complements are used in place of their corresponding binary numbers in some arithmetic computations in computers. Also called *one's complement*.

bit string: See *variable length bit field*.

block: (1) The smallest addressable unit of data that a device can transfer in an I/O operation (512 contiguous bytes for most disk devices). (2) An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

branch access type: An instruction attribute that indicates that the processor does not reference an operand address, but rather that the operand is a branch displacement. The size of the branch displacement is given by the data type of the operand.

branch mode: In branch address mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated program counter (which is the address of the first byte beyond the displacement), and the result is the branch address.

byte: Eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit.

cache memory: A small, high-speed memory placed between main memory and the processor.

call frame: *See stack frame.*

Call instructions: The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

call stack: The stack and conventional stack structure used during a procedure call. Each access mode of each process context has one call stack, and the interrupt service context has one call stack.

character: A symbol represented by an ASCII code. *See also alphanumeric character.*

character string: A contiguous set of bytes identified by two attributes—an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor: A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

compatibility mode: *See PDP-11 compatibility mode.*

condition codes: Four bits in the Processor Status Word (PSW) that indicate the results of previously executed instructions.

condition handler: A procedure that a process wants the system to execute when an exception condition occurs.

console: A manual-control unit integrated into the central processor that enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal: The part of a computer used by the operator to determine the status of, and to control, the operation of the computer. The console may have controls and indicators that are used for manual operation of the computer.

context indexing: The process of indexing through a data structure automatically because the size of the data type is known and is used to determine the offset factor.

context switching: Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution.

control region: The highest-addressed half of process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process.

control region base register (P1BR): The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

control region length register (P1LR): The processor register, or its equivalent in a hardware process control block, that contains the number of non-existent page table entries for virtual pages in a process control region.

current access mode: The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword (PSL) indicates the access mode of the currently executing software.

D__ floating datum: Eight contiguous bytes starting on an addressable byte boundary that are interpreted as containing a floating point number. The bits are labeled from right to left, 0 to 63.

data structure: Any table, list, array, queue, or tree whose format and access conventions are well defined for reference by one or more images.

data type: In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include floating point, integer, character string, packed decimal string, numeric string, queue, and variable length bit field.

descriptor: A data structure used in calling sequences for passing argument types, addresses and other optional information. *See character string descriptor.*

device interrupt: An interrupt received on interrupt priority levels 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device register: A location in device controller logic used to request device functions (such as I/O transfers) and/or to report status.

diagnostic: A program that tests logic and reports any faults it detects.

direct mapping cache: A cache organization in which any block of main memory data can be placed in only one possible position in the cache. *Compare with fully associative cache.*

displacement deferred indexed mode: An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode: A mode in which the specifier extension is a byte, word, or longword displacement. The displacement is sign-extended to 32 bits and added to a base address obtained from the specified registers. The result is the address of a longword that contains the address of the actual operand.

displacement indexed mode: A mode in which the base operand specifier uses displacement mode addressing.

displacement mode: A displacement addressing mode in which the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand.

double floating datum: *See D__ floating datum.*

effective address: The address obtained after indirect or indexing modifications are calculated.

entry mask: A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the *call* and *return* instructions.

entry point: A location that can be specified as the object of a *call* instruction. It contains an entry mask and exception enables known as the *entry point mask*.

event: A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process's ability to execute. Events can be synchronous with the process's execution (wait request), or they can be asynchronous (I/O completion).

exception: An event that changes the normal flow of instruction or set of instructions. Interrupts and *branch*, *call*, *case*, and *jump* instructions are excluded from this class of events. Exceptions are detected by the hardware. There are three types of hardware exceptions—traps, faults, and aborts.

exception condition: A hardware- or software-detected event other than an interrupt or *jump*, *branch*, *case*, or *call* instruction that changes the normal flow of instruction execution.

exception enables: *See trap enables.*

exception vector: *See vector.*

executive mode: The second most privileged processor access mode (mode 1). The Record Management Services (RMS) and many of the operating system's programmed service procedures execute in executive mode.

F__ floating datum: Four contiguous bytes starting on an addressable byte boundary. The bits are labeled from right to left 0 to 31. A two-word floating-point number is identified by the address of the byte containing bit 0.

fault: A hardware exception condition that occurs in the middle of an instruction. A fault leaves the registers and memory in a consistent state so the elimination of the fault and restarting the instruction gives correct results.

field: A set of contiguous bytes in a logical record. *See also variable length bit field.*

floating (point) datum: *See F__ floating datum.*

frame pointer: General register 13 (R13). By convention, the frame pointer (FP) contains the base address of the most recent call frame on the stack.

fully associative cache: A cache organization in which any block of data from main memory can be placed anywhere in the cache. *Compare with direct mapping cache.*

G__ floating datum: Eight contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 63. The address of a G__ floating datum is specified by the address of the byte containing bit 0.

general register: Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers that can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

giga: A prefix meaning 1,000,000,000 (10^9). In the computer industry, giga is often used to mean 1,073,741,824 (2^{30}) which is about 7.4 percent larger.

H__ floating datum: Sixteen contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from the right 0 through 127. The address of an H__ floating datum is specified by the address of the byte containing bit 0.

hardware context: The values contained in the following registers while a process is executing—the Program Counter (PC), the Processor Status Longword (PSL), the 14 general registers (R0 through R13), the four processor registers (POBR, POLR, P1BR, and P1LR), the Stack Pointer (SP) for the current access mode in which the processor is executing, and the contents to be loaded in the stack pointer for every access mode other than the current access mode. When a process is executing, its hardware context is continuously updated by the processor. When a process is not executing, its hardware context is stored in its hardware process control block.

hardware process control block (PCB): A data structure known to the processor that contains the hardware context when a process is not executing. A process's hardware PCB resides in its process header.

image file: A file containing the necessary information to establish an incarnation of a user program in a process including the memory image. Image files can be of the executable, shareable, and system types.

immediate mode: Autoincrement mode addressing in which the program counter is used as the register.

incarnation: A resource that is automatically allocated on a call or recursive call. A resource is a physical part of the computer such as a device, memory, or an interlocked data structure.

indexed addressing mode: In indexed mode addressing, two registers are used to determine the actual instruction operand — an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array.

index register: A register containing an address offset.

instruction buffer: An 8-byte buffer in the processor used to contain bytes of the instruction currently being decoded and to prefetch instructions in the instruction stream. The control logic continuously fetches data from memory to keep the 8-byte buffer full.

interleaving: Assigning consecutive physical memory addresses alternately between two memory controllers.

internal processor register: A part of the processor used by the operating system software to control the execution states of the computer system. Sometimes called privileged processor register. These registers are accessed with MTPR and MFPR instructions.

interrupt: An event other than an exception or *branch*, *call*, *case*, or *jump* instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also **device interrupt**, **software interrupt**, and **urgent interrupt**.

interrupt priority level (IPL): The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels (IPL). IPL 1 is lowest, 31 highest.

interrupt service routine: The software executed when a device interrupt occurs.

interrupt stack: The systemwide stack used when executing in interrupt service context. At any time, the processor is either in a process context or in systemwide interrupt service context. When executing in interrupt service context, the processor is operating with kernel privileges on the kernel or interrupt stack. The interrupt stack is not context-switched or swapped.

interrupt stack pointer: The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

interrupt vector: *See vector.*

kernel mode: The most privileged processor access mode (mode 0). The operating system's most privileged services (I/O drivers, the pager) run in kernel mode.

literal mode: An addressing mode in which the instruction operand is a constant whose value is expressed in a 6-bit field of the instruction.

longword: Four contiguous bytes starting on an addressable byte boundary. Bits are numbered from right to left 0 through 31.

main memory: *See physical memory.*

mass-storage device: A device capable of reading and writing data on mass storage media such as a diskpack or a magnetic tape reel.

memory management: The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

Memory Mapping Enable (MME): A bit in a processor register that governs address translation.

modify access type: A specific way of accessing characterized by a specified operand of an instruction or procedure being read, and potentially modified and written, during that instruction's or procedure's execution.

native mode: *See VAX native mode.*

nibble: Four bits of memory; one half of a byte.

normalized fraction: A numeric representation patterned on scientific notation, but in which the fraction part of the representation is greater than or equal to 0.5 and less than 1. As a binary form, such a fraction always begins with a 1 in the leftmost (most significant) bit, unless the number is zero. Because of this, the lead 1 is not stored, and a bit-per-number saving is effected in storage.

numeric string: A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

octaword: An octaword is 16 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from the right 0 through 127. An octaword is specified by the address of the byte containing bit 0.

offset: A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

one's complement: *See bit complement.*

opcode: Short form of operation code. That part of a machine language instruction that identifies the operation the CPU is to perform. The pattern of bits within an instruction that specifies the operation to be performed.

operand specifier: The pattern of bits in an instruction that indicates the addressing mode and a register or displacement that identifies an instruction operand.

operand specifier type: The access type and data type of an instruction's operand(s). For example, test instructions are of read access type because they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

packed decimal: A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers 0 through 9.

packed decimal string: A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit, except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page: (1) A set of 512 contiguous byte locations used as the unit of memory mapping and protection. (2) The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

page fault: An exception generated by a reference to a page that is not mapped into a working set.

page fault cluster size: The number of pages read in on a page fault.

page frame number (PFN): The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page table entry (PTE): The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page. When it is not in memory, the PTE contains the information needed to locate the page on secondary storage (disk).

paging: The process of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages need to reside in physical memory. The remaining pages can reside on disk until they are needed in physical memory. In this system, a process is paged only when it references more pages than it is allowed to have in its working set. A page fault occurs when the process refers to a page not in its working set. This causes the operating system's pager to read in the referenced page if it is on disk, replacing the least recently faulted pages as needed. A process pages only against itself; that is, one process cannot exceed the working set limit assigned to it by bringing in more than its quota of pages. This protects other processes in the system.

PDP-11 compatibility mode: (This is now an optional feature of the VAX architecture.) A mode of execution that enables the central processor to execute nonprivileged PDP-11 instructions.

physical address: The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space: The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical memory: The memory modules connected to the Synchronous Backplane Interconnect that are used to store (1) instructions that the processor can directly fetch and execute, and (2) any other data that a processor is instructed to manipulate. Also called *main memory*.

position dependent code: Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

position independent code: Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addressing modes that form an effective address relative to the Program Counter register.

privileged instruction: In general, any instruction intended for use by the operating system or privileged system programs. In particular, an instruction that the processor does not execute unless the current access mode is kernel mode (for example, HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

privileged processor register: See **internal processor register**.

procedure: A routine entered by way of a *call* instruction.

process: The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software contexts.

process address space: See **process space**.

process context: The hardware and software contexts of a process.

process control block (PCB): A data structure used to contain the process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB. See also **hardware process control block**.

process page tables: The page tables used to describe process virtual memory.

process space: The lowest-addressed half of virtual address space, where process instructions and data reside. Process space is divided into a program region and a control region.

Processor Status Longword (PSL): A system-programmed processor register consisting of a word of privileged processor status and the PSW. The privileged processor status information includes the current interrupt priority level, the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

Processor Status Word (PSW): The low-order word of the Processor Status Longword. Processor status information includes the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

Program Counter (PC): General register 15 (R15). At the beginning of an instruction's execution, the program counter (PC) normally contains the address of a location in memory from which the processor will fetch the next instruction to execute.

program locality: An indication of the proximity of a program's references to virtual memory locations. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

program region: The lowest-addressed half of process address space (P0 space). The program region contains the image being executed by the process and other user code called by the image.

program region base register (P0BR): The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of the page table entry for virtual page number 0 in a process program region.

program region length register (P0LR): The processor register, or its equivalent in a hardware process control block, that contains the number of entries in the page table for a process program region.

quadword: Eight contiguous bytes (64 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 63. A quadword is identified by the address of the byte containing the low-order bit (bit 0).

queue: (1) *noun*. A circular, doubly linked list. (2) *verb*. To make an entry in a list or table, perhaps using the INSQUE instruction.

read access type: An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

register: A storage location in hardware logic other than main memory. *See also* **general register, processor register, device register**.

register deferred indexed mode: An indexed addressing mode in which the base operand specifier uses register deferred mode addressing.

register deferred mode: An addressing mode in which the contents of the specified register are used as the address of the actual instruction operand.

register mode: An addressing mode in which the contents of the specified register are used as the actual instruction operand.

scatter/gather: A method used to transfer in one I/O operation data from discontinuous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontinuous pages in memory.

secondary storage: Random access mass storage.

signal: (1) An electrical impulse conveying information. (2) The software mechanism used to indicate that an exception condition was detected.

software interrupt: An interrupt generated on interrupt priority levels 1 through 15, that can be requested only by software.

software process control block: *See* **process control block**.

stack: An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in/ first-out concept. As items are added to (pushed on) the stack, the stack pointer decrements. As items are retrieved from (popped off) the stack, the stack pointer increments.

stack frame: A standard data structure built on the stack during a procedure call, starting from the location addressed by the frame pointer and going to lower addresses, and popped off during a return from procedure. Also called *call frame*.

Stack Pointer (SP): General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers—kernel, executive, supervisor, user, or interrupt—depending on the value in the current mode and interrupt stack bits in the processor status longword.

store through: *See* **write through**.

Supervisor mode: The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

Synchronous Backplane Interconnect (SBI): That part of the hardware that interconnects the processor, memory controllers, MASSBUS adapters, and the UNIBUS adapter.

system address space: *See system space, system region.*

system base register (SBR): A processor register that contains the physical address of the base of the system page table.

system control block (SCB): The data structure in system space that contains all the interrupt and exception vectors known to the system.

system control block base register (SCBB): A processor register containing the base address of the system control block.

system identification register (SIR): A processor register which contains the processor type and serial number.

system length register (SLR): A processor register containing the length of the system page table in longwords, that is, the number of page table entries in the system region page table.

system page table (SPT): The data structure that maps the system region virtual addresses, including the addresses used to refer to the process page tables. The system page table (SPT) contains one page table entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the system base register (SBR).

system region: The third quarter of virtual address space; that is, the lower-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are system entry vectors, the system control block (SCB), the system page table (SPT), and process page tables.

system space: The higher-addressed half of virtual address space. *See also system region.*

system virtual address: A virtual address identifying a location mapped by an address in system space.

system virtual space: *See system space.*

terminal: The general name for those peripheral devices that have keyboards and videoscreens or printers. Under program control, a terminal enables users to type commands and data on the keyboard and receive messages on the videoscreen or printer. Examples of terminals are the LA38 DEC-writer hardcopy terminal and the VT100 video display terminal.

translation buffer: An internal processor cache containing translations for recently used virtual addresses.

trap: An exception condition that occurs at the end of the instruction that caused the exception. The program counter (PC) saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

trap enables: Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

two's complement: A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

two-way associative cache: A type of cache memory organization that has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into either group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations. It takes advantage of the features of both.

undefined: An operation that may vary from moment to moment, implementation to implementation, and instruction to instruction. The operation can vary in effect from doing nothing to halting system operation. Nonprivileged software should avoid invoking operations identified as *undefined*.

unpredictable: Results of an operation that may vary from moment to moment, implementation to implementation, and instruction to instruction. Engineering Change Orders (ECOs) may alter *unpredictable* results. Software should not depend on results specified as *unpredictable*.

urgent interrupt: An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and powerfail.

user mode: The least privileged processor access mode (mode 3). User processes and the Runtime Library procedures run in user mode.

user privileges: The privileges granted a user by the system manager.

variable length bit field: A set of 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field has four attributes—the address of a byte, the bit position of the starting location of the bit field with respect to bit 0 of the byte address, the size of the bit field in bits, and whether the field is signed or unsigned.

VAX native mode: The processor's primary execution mode.

vector: (1) An interrupt or exception vector is a storage location, known to the system, that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. (2) For the purposes of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. (3) A one-dimensional array.

virtual address: A 32-bit integer identifying a byte *location* in virtual address space. The memory management hardware translates a virtual address to a physical address. The term *virtual address* may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space: The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or of data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 (2^{32}) byte addresses.

virtual memory: The set of storage locations in physical memory and on disk that is referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for nonresident virtual memory.

virtual page number: The virtual address of a page of virtual memory.

word: Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing bit 0.

working set: The set of pages in process address space to which an executing process can refer without incurring a page fault. The working set must be resident in memory for the process to execute. The remaining pages of that process, if any, are either in memory and not in the process working set or they are on secondary storage.

write access type: The specified operand of an instruction or procedure that is only written during that instruction's execution.

write allocate: A cache management in which cache is allocated on a write miss as well as on the usual read miss.

write back: A cache management technique in which data from a write operation to cache is copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. *Compare with* **write through**.

write through: A cache management technique in which data from a write operation is copied in both cache and main memory. Cache and main memory data are always consistent. *Compare with* **write back**.

Index

A

aborts, 8-1—8-2
 caused by kernel stack not valid
 exceptions, 8-14
 caused by reserved operand
 exceptions, 8-13
absolute index mode, 5-25—5-26
absolute mode, 2-6, 5-37
absolute queues, 2-13, 4-17, 4-18
 instructions for, 6-19—6-22
access
 control of, 7-16—7-19
 to privileged system services,
 7-22—7-23
 to shared data, synchronization of,
 3-3—3-4
 to stack registers, 8-30
access control violation faults, 7-20,
 8-12, 8-32
access modes, 2-5, 3-1—3-2, 7-17
 asynchronous system traps and, 8-8
 change mode instruction for,
 9-14—9-15
 in PDP-11 compatibility mode, 11-2
 privileged instructions for,
 6-15—6-16
access types, 2-7
 in operand notation, 5-1—5-2
add instructions
 add, 6-2, 6-11, 6-12, 9-1
 add aligned word interlocked
 (ADAWI), 3-3, 6-13, 9-2
 add compare and branch (ACB),
 6-4, 9-2—9-3
 add one and branch (AOB), 6-4, 9-3
 add packed (ADDP), 6-6, 9-3—9-4
 add with carry (ADWC), 6-13, 9-4

address and branch access, 2-7

addresses

 instructions for, 6-1—6-2
 move address instruction for, 9-44
 in PDP-11 compatibility mode, 11-7
 push address instruction for, 9-53
 translation of, 7-8—7-16
 virtual, 2-4
 virtual address extension for,
 1-2—1-3
 in virtual address space, 1-4

addressing

 of general registers, 2-9
 virtual, 2-4

addressing modes, 2-6, 2-9, 5-10—5-12

 absolute, 5-37
 absolute index, 5-25—5-26
 autodecrement, 5-12—5-13
 autodecrement index, 5-24—5-25
 autoincrement, 5-14
 autoincrement deferred, 5-14—5-16
 autoincrement deferred index,
 5-23—5-24
 autoincrement index, 5-22—5-23
 branch mode addressing,
 5-41—5-43
 displacement, 5-17
 displacement deferred, 5-17—5-19
 displacement deferred index,
 5-27—5-28
 displacement index, 5-36—5-27
 faults and, 8-12
 general mode addressing, 5-12
 general register addressing, 5-12
 immediate, 5-37—5-39
 index, 5-19—5-21
 literal, 5-29—5-32
 program counter register
 addressing, 5-36—5-37
 register, 5-32—5-33

addressing modes, (*cont.*)

- register deferred, 5-33—5-35
- register deferred index, 5-21—5-22
- relative, 5-39—5-40
- relative deferred, 5-41
- restarts and, 3-18
- address space, virtual, 7-6—7-8
- address translation maps, 7-7
- architectural subsetting, 10-1—10-4
- architecture, 1-1—1-2
 - of PDP-11 compatibility mode, 11-2
- argument lists, 3-5
- argument pointer (AP), 3-5
- arithmetic exceptions, 8-9—8-11
- arithmetic instructions, 6-2
 - add, 9-1
 - arithmetic shift and round packed (ASHP), 6-6, 9-4—9-5
 - arithmetic shift (ASH), 6-13, 9-4
 - divide, 9-26—9-27
 - multiply, 9-50—9-51
 - subtract, 9-60—9-61
- array processing
 - autoincrement mode used for, 5-14
 - index instruction for, 6-12
 - index mode used for, 5-19
- assembler, notation conventions for, 5-1
- ASTLVL registers, 8-8
- asynchronous system traps (ASTs), 2-3, 8-7—8-9
- autodecrement index mode, 5-24—5-25
- autodecrement mode, 5-12—5-13
- autoincrement deferred index mode, 5-23—5-24
- autoincrement deferred mode, 2-6, 5-14—5-16
- autoincrement index mode, 5-22—5-23
- autoincrement mode, 5-14

B

- backward link, 4-17
- base addresses, 6-25
- batteries, for time-of-year clock, 3-7
- binary normalized numbers, 6-10
- bit clear (BIC) instruction, 6-13, 9-5
- bit clear processor status word (BICPSW) instruction, 6-18, 9-5
- bit set (BIS) instruction, 6-13, 9-6
- bit set processor status word (BISPSW) instruction, 6-18, 9-6
- bit test (BIT) instruction, 6-14, 9-6—9-7
- branch instructions, 6-4—6-5
 - add compare and branch, 9-2—9-3
 - add one and branch, 9-3
 - branch (BR), 9-7
 - branch on bit (BB), 9-7
 - branch on bit and modify without interlock (BR), 9-8
 - branch on bit and clear and clear, interlocked (BBCCI), 3-3
 - branch on bit interlocked (BB), 9-7—9-8
 - branch on bit set and set, interlocked (BBSSI), 3-3
 - branch on condition, 9-8—9-10
 - branch on low bit (BLB), 9-10—9-11
 - branch to subroutine (BSB), 2-7, 6-4, 9-11
 - case, 9-14
 - subtract one and branch, 9-61
- branch mode addressing, 5-41—5-43
- breakpoint fault (BPT) instruction, 6-5, 9-11
- breakpoint faults, 8-11
- budcheck (BUG) instruction, 9-11—9-12
- byte data type, 4-7
- bytes, 4-7, 7-3

C

- cache memory, 3-16—3-17
- call frames (stack frames), 2-7, 3-5, 6-14, 6-17
- call instructions, 2-7, 3-2, 6-4—6-5, 6-16—18
 - to access privileged system services, 7-22
- call procedure with general argument list (CALLG), 6-16, 9-12
- call procedure with stack argument lists (CALLS), 9-13
- call procedure with stack pointer argument list (CALLS), 6-16, 6-17
 - extended function call, 9-35
 - frame pointer and, 3-5
 - trace exceptions and, 8-17
- carry condition code, 2-13, 8-5
 - case instructions, 6-4, 9-14
 - change mode (CHM) instruction, 3-2, 6-15, 7-22, 9-14—9-15
- character string data type, 2-12, 4-1—4-2
 - edit instruction to convert packed decimal to, 6-9
- character string instructions, 6-2—6-3, 10-2
 - compare characters, 9-16—9-17
 - edit, 9-27—9-34
 - locate character, 9-42—9-43
 - match character, 9-43
 - move character, 9-44—9-45
 - move translated characters, 9-49
 - move translated until character, 9-49
 - scan characters, 9-59
 - skip characters, 9-59—9-60
 - span characters, 9-60
- clear (CLR) instruction, 6-12, 6-14, 9-15
- clock registers, 3-7—3-9
 - comments, in MACRO source statements, 5-5
- compare characters (CMPC) instruction, 6-2, 9-16—9-17
- compare field (CMP) instruction, 6-25, 9-17
- compare (CMP) instruction, 6-12, 6-14, 9-15—9-16
- compare packed (CMPP) instruction, 6-6, 9-17
- compatibility, with PDP-11 systems, 1-3, 11-1—11-11
- compatibility mode bit, 8-7
- compatibility mode faults, 8-12
- compatibility mode instruction set, 10-2
- computation instructions, 6-1
- computers, architecture of, 1-1
- condition codes, 2-13—2-14, 5-5, 8-5
- console receive control/status register, 3-9
- console registers, 10-3
- console terminal registers, 3-9
- console transmit control/status register, 3-9
- constants, literal mode for, 5-29—5-30
- context load operations, 3-7
- contexts, 2-1, 3-2
 - for exceptions and interrupts, 2-14
 - during exceptions and interrupts, 8-4
 - process, 8-27
 - switched, 1-3, 2-2—2-3, 3-7
- context save operations, 3-7
- control instructions, 6-3—6-5
 - case, 9-14
 - jump, 9-41
 - jump to subroutine, 9-41—9-42
 - See also* branch instructions; subroutines

conversion exceptions, 8-9
 conversion instructions, 6-1
 convert (CVT) instruction, 6-12, 6-14,
 9-18—9-20
 convert leading separate numeric to
 packed (CVTSP) instruction, 6-6,
 9-20
 convert longword to packed (CVTLP)
 instruction, 6-6, 9-20—9-21
 convert packed to leading separate
 numeric (CVTPS) instruction, 6-6,
 9-21
 convert packed to longword (CVTPL)
 instruction, 6-6, 9-21
 convert packed to trailing numeric
 (CVTPT) instruction, 6-6, 9-22
 convert rounded (CVTR) instruction,
 6-12
 convert trailing numeric to packed
 (CVTTP) instruction, 6-6, 9-23
 cyclic redundancy check (CRC)
 instruction, 6-5—6-6, 9-23—9-25

D

data
 in registers, 4-24—4-25
 sharing of, 3-3—3-4
 data types, 2-10—2-13, 4-1
 address instructions for, 6-2
 character string, 4-1—4-2
 convert instructions for, 9-18—9-23
 floating-point, 4-2—4-6
 floating-point, instructions for,
 10-1—10-2
 instructions symmetrical with, 2-5,
 6-1
 integer, 4-6—4-9
 numeric string, 4-10—4-15
 in operand notation, 5-2
 packed decimal string, 4-16—4-17
 queue, 4-17—4-21
 variable length bit field, 4-21—4-24

debugging, trace exceptions during,
 8-15
 decimal overflow trap enable bit, 8-6
 decimal string data types
 packed, 2-12
 packed decimal string, 4-16—4-17
 decimal string divide by zero trap
 exceptions, 8-10
 decimal string instructions, 6-6—6-9,
 10-2
 add packed, 9-3—9-4
 arithmetic shift and round packed,
 9-4—9-5
 compare packed, 9-17
 convert leading separate numeric to
 packed, 9-20
 convert longword to packed,
 9-20—9-21
 convert packed to leading separate
 numeric, 9-21
 convert packed to longword, 9-21
 convert packed to trailing numeric,
 9-22
 convert trailing numeric to packed,
 9-23
 divide packed, 9-27
 move packed, 9-48
 multiply packed, 9-51
 subtract packed, 9-62
 decimal string overflow trap
 exceptions, 8-10
 decrement (DEC) instruction, 6-13,
 9-26
 device controllers, interrupt vectors in,
 8-19
 device interrupts, 8-19
 D floating-point data type, 2-12,
 4-2—4-3, 6-10
 instructions for, 10-1
 stored in registers, 4-24, 4-25
 displacement deferred index mode,
 5-27—5-28
 displacement deferred mode, 2-6,
 5-17—5-19
 displacement index mode, 5-26—5-27

displacement mode, 2-6, 5-17
 divide by zero floating fault exceptions,
 8-11
 divide (DIV) instruction, 6-2,
 6-10—6-12, 9-26—9-27
 divide packed (DIVP) instruction, 6-6,
 9-27
 documentation, 1-14
 double-precision floating (D_) data
 type, 4-2

E

edit (EDITPC) instruction, 6-9—6-10,
 9-27—9-34
 emulation
 of instructions, 10-4
 of PDP-11 user environment,
 11-2—11-6
 environments
 for PDP-11 compatibility mode,
 11-2—11-6
 for programming, 3-1—3-4
 EO\$ADJUST__INPUT operator, 9-29
 EO\$BLANK__ZERO operator, 9-29
 EO\$CLEAR__SIGNIF operator,
 9-29—9-30
 EO\$END__FLOAT operator, 9-30
 EO\$END operator, 9-30
 EO\$FILL operator, 9-30
 EO\$FLOAT operator, 9-31
 EO\$INSERT operator, 9-31—9-32
 EO\$LOAD operator, 9-32
 EO\$MOVE operator, 9-32—9-33
 EO\$REPLACE__SIGN operator, 9-33
 EO\$SET__SIGNIF operator,
 9-33—9-34
 EO\$STORE__SIGN operator, 9-34
 errors
 cyclic redundancy check instruction
 for, 6-5—6-6
 processor, 3-18
 in use of stacks, 3-14
 event handling, 8-1—8-2
 asynchronous system traps for,
 8-7—8-9
 for exceptions and interrupts,
 8-3—8-4
 interrupt priority levels for, 8-3
 processor status and, 8-4—8-7
 system control block vectors for,
 8-23—8-27
 See also exceptions; interrupts
 exceptions, 2-3, 2-14, 8-1, 8-3—8-4
 arithmetic, 8-9—8-11
 event handling of, 8-1—8-2
 initiating, 8-31—8-33
 instruction faults, 8-11—8-12
 memory management, 8-12
 operand reference, 8-12—8-14
 in PDP-11 compatibility mode,
 11-10
 priority of recognition of,
 8-30—8-31
 processor status during, 8-4—8-7
 return from exception or interrupt
 instruction for, 9-56—9-57
 serious system failures, 8-14—8-15
 system control block and,
 8-23—8-27
 trace, 8-15—8-17
 exclusive OR (XOR) instruction, 6-14,
 9-34
 executive access mode, 2-5, 3-2, 7-1
 executive stack pointer (ESP), 8-27
 extended divide (EDIV) instruction,
 6-13, 9-35
 extended function call (XFC)
 instruction, 6-15, 9-35
 extended modulus (EMOD)
 instruction, 6-12, 9-35—9-36
 extended multiply (EMUL) instruction,
 6-13, 9-36

extent notation, 5-5
 extract field (EXT) instruction, 6-25,
 9-36—9-37

F

failures, system, 8-14—8-15
 fault parameter word, 7-20—7-21
 faults, 8-2
 instruction, 8-11—8-12
 kernel stack and, 8-28
 memory management, 7-20—7-21,
 8-12
 operand reference, 8-12—8-14
 trace, in PDP-11 compatibility
 mode, 11-10
 See also exceptions
 F floating-point data type, 2-12,
 4-3—4-4, 6-10
 instructions for, 10-1
 stored in registers, 4-24

field access, 2-7

fields
 instructions for, 6-25
 variable-length bit field data type,
 2-12, 4-21—4-24
 See also variable-length bit field
 instructions

find first bit (FF) instruction, 6-25,
 9-37

flags, trap-enable, 8-5

floating-overflow fault exceptions, 8-11

floating-point data types, 2-12,
 4-2—4-6

floating-point instructions, 6-10—6-12,
 10-1—10-2
 add, 9-1
 clear, 9-15
 compare, 9-15—9-16
 convert, 9-18—9-20
 divide, 9-26—9-27
 extended modulus, 9-35—9-36
 move, 9-43—9-44

floating-point instructions, (*cont.*)

 move negated, 9-47—9-48
 multiply, 9-50—9-51
 polynomial evaluation, 9-51—9-52
 subtract, 9-60—9-61
 test, 9-63

floating-point literals, 5-30

floating underflow enable bit, 6-17

floating-underflow exception enable
 bit, 8-6

floating-underflow fault exceptions,
 8-11

format
 for instructions, 5-5—5-10
 for MACRO source statements,
 5-5—5-6

forward link, 4-17

frame pointer (FP), 3-5

G

general mode addressing, 5-12

 general register addressing,
 5-12—5-35

 program counter register
 addressing, 5-36—5-41

general register addressing, 5-12

 absolute index mode, 5-25—5-26
 autodecrement index mode,
 5-24—5-25

 autodecrement mode, 5-12—5-13
 autoincrement deferred index
 mode, 5-23—5-24

 autoincrement deferred mode,
 5-14—5-16

 autoincrement index mode,
 5-22—5-23

 autoincrement mode, 5-14
 displacement deferred index mode,
 5-27—5-28

 displacement deferred mode,
 5-17—5-19

 displacement index mode,
 5-26—5-27

general register addressing, (*cont.*)

- displacement mode, 5-17
- index mode, 5-19—5-21
- literal mode, 5-29—5-32
- register deferred index mode,
5-21—5-22
- register deferred mode, 5-33—5-35
- register mode, 5-32—5-33

general registers, 3-4—3-6

- addressing of, 2-9
- for PDP-11 compatibility mode,
11-2—11-3

G_floating-point data type, 2-12,

- 4-4—4-5, 6-10
- instructions for, 10-2

global page table base register (GBR),
7-11

global page table index (GPTX), 7-10

H

halt instruction, 6-15, 9-37—9-38

handbooks, 1-4

handling routines, 8-17

hardware

- high-level language instructions
implemented in, 1-2
- interrupt priority levels for, 8-3
- interrupts generated by, 8-19
- memory management, 1-3—1-4,
2-4, 7-1, 7-19, 7-20
- for multiprogramming, 2-1
- page table entries for, 7-10—7-11
- for PDP-11 compatibility mode,
11-2

hardware context, 2-1, 2-2

hardware process control block, 2-2

head of the queue, 4-18, 6-19

hexadecimal numbers, assembler
notation for, 5-1

H_floating-point data type, 2-12,
4-5—4-6, 6-10

H_floating-point data type, (*cont.*)

- instructions for, 10-2
- stored in registers, 4-25

high-level languages, 1-2

I

immediate mode, 2-6, 5-37—5-39

increment (INC) instruction, 6-13, 9-38

index instruction, 6-12, 9-38

index mode, 5-19—5-21

index registers, 5-19

input/output control, 2-15

input/output device controllers, 2-15

input/output devices, page table entries
for, 7-10—7-11

input/output references in PDP-11
compatibility mode, 11-11

input/output registers, 3-13—3-14

insert entry in queue (INSQUE)
instruction, 6-22, 9-38—9-39

insert entry into queue at head,
interlocked (INSQHI) instruction,
3-4, 6-24, 9-39—9-40

insert entry into queue at tail,
interlocked (INSQTI) instruction,
3-4, 6-24, 9-40

insert field (INSV) instruction, 6-25,
9-41

instruction faults, 8-11—8-12

instructions and instruction sets, 1-2,
2-5—2-6

aborts caused by, 8-1

access modes for, 3-1

add, 9-1

add aligned word interlocked, 9-2

add compare and branch, 9-2—9-3

add one and branch, 9-3

add packed, 9-3—9-4

address, 6-1—6-2

addressing modes and, 5-10—5-43

instructions and instruction sets, (*cont.*)

- add with carry, 9-4
- arithmetic, 6-2
- arithmetic shift, 9-4
- arithmetic shift and round packed, 9-4—9-5
- bit clear, 9-5
- bit clear processor status word, 9-5
- bit set, 9-6
- bit set processor status word, 9-6
- bit test, 9-6—9-7
- branch, 9-7
- branch on bit, 9-7
- branch on bit and modify without interlock, 9-8
- branch on bit interlocked, 9-7—9-8
- branch on condition, 9-8—9-10
- branch on low bit, 9-10—9-11
- branch to subroutine, 9-11
- breakpoint fault, 9-11
- bugcheck, 9-11—9-12
- call procedure with general argument list, 9-12
- call procedure with stack argument list, 9-13
- case, 9-14
- to change access mode, 7-22—7-23
- change mode, 9-14—9-15
- character string, 6-2—6-3
- clear, 9-15
- compare, 9-15—9-16
- compare characters, 9-16—9-17
- compare field, 9-17
- compare packed, 9-17
- compatibility mode instruction set, 10-2
- condition codes for, 2-13—2-14
- control, 6-3—6-5
- convert, 9-18—9-20
- convert leading separate numeric to packed, 9-20
- convert longword to packed, 9-20—9-21
- convert packed to leading separate numeric, 9-21
- convert packed to longword, 9-21
- convert packed to trailing numeric, 9-22

instructions and instruction sets, (*cont.*)

- convert trailing numeric to packed, 9-23
- cyclic redundancy check, 6-5—6-6, 9-23—9-25
- data types recognized by, 2-10—2-13
- decimal string, 6-6—6-9
- decrement, 9-26
- divide, 9-26—9-27
- divide packed, 9-27
- edit, 6-9—6-10, 9-27—9-34
- emulation of, 10-4
- exclusive OR, 9-34
- extended divide, 9-35
- extended function call, 9-35
- extended modulus, 9-35—9-36
- extended multiply, 9-36
- extract field, 9-36—9-37
- faults during execution of, 8-2
- find first bit, 9-37
- floating-point, 6-10—6-12, 10-1—10-2
- floating-point instructions, 10-1—10-2
- format for, 5-6—5-10
- halt, 9-37—9-38
- increment, 9-38
- index, 6-12, 9-38
- insert entry in queue, 9-38—9-39
- insert entry in queue at head, interlocked, 9-39—9-40
- insert entry in queue at tail, interlocked, 9-40
- insert field, 9-41
- integer, 6-13
- jump, 9-41
- jump to subroutine, 9-41—9-42
- kernel instruction set, 10-3—10-4
- load process context, 9-42
- locate character, 9-42—9-43
- logic, 6-13—6-14
- MACRO source statement format for, 5-5—5-6
- match characters, 9-43
- on MicroVAX I and II systems, 10-4
- move, 9-43—9-44

instructions and instruction sets, (*cont.*)

- move address, 9-44
- move characters, 9-44—9-45
- move complement, 9-45
- move from processor register, 9-45—9-47
- move from processor status longword, 9-47
- move negated, 9-47—9-48
- move packed, 9-48
- move to processor register, 9-48
- move translated characters, 9-49
- move translated until characters, 9-49
- move zero-extended, 9-50
- multiple register, 6-14
- multiply, 9-50—9-51
- multiply packed, 9-51
- notation conventions for, 5-1—5-5
- operand processing by, 2-7—2-8
- for PDP-11 compatibility mode, 11-4—11-6
- polynomial evaluation, 9-51—9-52
- pop registers, 9-52
- privileged, 6-15—6-16
- probe accessibility, 9-52—9-53
- procedure call, 6-16—6-18
- process control, 2-8
- processor status longword, 6-18
- push address, 9-53
- push longword, 9-54
- push registers, 9-54
- queue, 6-19—6-24
- remove entry from queue, 9-54—9-55
- remove entry from queue at head, interlocked, 9-55—9-56
- remove entry from queue at tail, interlocked, 9-56
- restarts and, 3-17—3-18
- return from exception or interrupt, 9-56—9-57
- return from procedure, 9-57
- return from subroutine, 9-58
- rotate longword, 9-58
- routine calls, 2-7
 - save process context, 9-58—9-59
 - scan characters, 9-59

instructions and instruction sets, (*cont.*)

- for shared data, 3-3—3-4
- skip character, 9-59—9-60
- span characters, 9-60
- special, 3-2—3-3
- stacks and, 3-14
- string instructions, 10-2
- subtract, 9-60—9-61
- subtract one and branch, 9-61
- subtract packed, 9-62
- subtract with carry, 9-62
- suspended, 8-31
- test, 9-63
- trace exceptions between executions of, 8-15—8-17
- variable length bit field, 6-25
- integer data types, 2-12, 4-6—4-9
- integer divide by zero trap exceptions, 8-10
- integer instructions, 6-31
 - add aligned word interlocked, 9-2
 - add with carry, 9-4
 - decrement, 9-26
 - extended divide, 9-35
 - extended multiply, 9-36
 - increment, 9-38
 - push longword, 9-54
 - subtract with carry, 9-62
- integer overflow trap enable bit, 8-6
- integer overflow trap exceptions, 8-10
- internal (processor) registers, 3-7
- interrupt context, 3-2
- interrupt priority level register, 8-21—8-22
- interrupt priority levels (IPLs), 8-1, 8-3
 - asynchronous system traps and, 8-9
 - in PDP-11 compatibility mode, 11-10
 - processors', 8-6
 - for software interrupts, 8-19, 8-20
 - for urgent interrupts, 8-21
- interrupts, 2-3, 2-14, 3-18, 8-1, 8-3—8-4, 8-18—8-19
 - during character string instruction executions, 6-3

interrupts, (*cont.*)

- device, 8-19
- event handling of, 8-2
- example of, 8-22
- initiating, 8-31—8-33
- interrupt priority level register for, 8-21—8-22
- in PDP-11 compatibility mode, 11-10
- priority of recognition of, 8-30—8-31
- processor status during, 8-4—8-7
- restarts after, 3-17
- return from exception or interrupt instruction for, 9-56—9-57
- software-generated, 8-19—8-21
- system control block and, 8-23—8-27
- urgent, 8-21
- interrupt stack, 3-15, 8-27
 - in PDP-11 compatibility mode, 11-10
 - process scheduling software executed on, 2-8
- interrupt stack flag, 8-7
- interrupt stack not valid—halt exceptions, 8-14
- interval clock, 3-7—3-9
- interval clock control/status register, 3-8
- interval count register, 3-8
- interval timer registers, 10-2
- I/O references, *see* input/output references

J

- journals (of procedure call nesting), 2-7
- jump instructions, 6-4
- jump (JMP) instruction, 6-5, 9-41
- jump to subroutine (JSB) instruction, 2-7, 6-5, 9-41—9-42

K

- kernel access mode, 2-5, 3-2, 7-1
 - interrupt priority changed in, 8-2
 - process page table entries in, 7-8
- kernel instruction set, 10-3—10-4
- kernel reads, 7-14
- kernel stack, 8-28, 8-32
- kernel stack frame, 11-10
- kernel stack not valid—abort exceptions, 8-14, 8-32
- kernel stack pointer (KSP), 8-27

L

- labels, in MACRO source statements, 5-5
- languages, 1-2
- last-in/first-out (LIFO) queues (stacks), 3-14
- leading numeric string data type, 6-6
- leading separate numeric string data type, 4-10—4-12
- length registers, 7-9
- literal mode, 5-29—5-32
- literature, 1-4
- load process context (LDPCTX) instruction, 2-2, 3-3, 6-15, 7-22, 7-23, 9-42
 - stack pointers referenced by, 8-28, 8-33
 - translation buffer updated by, 7-20
- locate character (LOCC) instruction, 6-2, 9-42—9-43
- logical complement operations, 6-13
- logic instructions, 6-13—6-14
 - arithmetic shift, 9-4
 - bit clear, 9-5
 - bit set, 9-6
 - bit test, 9-6—9-7
 - clear, 9-15

logic instructions, (*cont.*)

- compare, 9-15—9-16
- convert, 9-18—9-20
- exclusive OR, 9-34
- move, 9-43—9-44
- move complement, 9-45
- move negated, 9-47—9-48
- move zero-extended, 9-50
- rotate longword, 9-58
- test, 9-63

longword data type, 4-8, 4-24

longwords, 7-3

loop control instructions, 6-4

M

machine checks, 8-4, 8-15, 8-21

MACRO source statements, 5-5—5-6

manuals, 1-4

map enable register (MAPEN), 7-19

match characters (MATCHC)
instruction, 6-2, 9-43

memory

- access modes for, 3-1
- cache, 3-16—3-17
- paging of, 7-1
- virtual, 7-2—7-6

memory management, 1-3—1-4,

2-4—2-5, 7-1—7-2

access control in, 7-16—7-19

access privileged system services
and, 7-22—7-23

address translation in, 7-8—7-16

control of, 7-19—7-20

exceptions in, 8-12

faults and parameters for,
7-20—7-21

interrupts and, 8-19

in PDP-11 compatibility mode,
11-7—11-9

virtual address space in, 7-6—7-8

virtual memory in, 7-2—7-6

memory mapping, 2-4

Memory Mapping Enable (MME) bit,
7-8

MicroVAX I systems, 10-4

MicroVAX II systems, 10-4

modify access, 2-7

move address (MOVA) instruction, 6-1,
9-44

move characters (MOVC) instruction,
6-2, 9-44—9-45

move complement (MCOM)
instruction, 6-14, 9-45

move from processor register (MFPR)
instruction, 6-15, 7-22, 9-45—9-47
processor registers and, 3-7, 7-23
process space address translation
and, 7-14, 7-16
to read map enable register, 7-19
for software interrupt summary
register, 8-20
stack pointers referenced by, 3-6,
8-33

move from processor status longword
(MOVPSL) instruction, 6-18, 9-47

move (MOV) instruction, 6-12, 6-14,
9-43—9-44

move negated (MNEG) instruction,
6-12, 6-14, 9-47—9-48

move packed (MOVP) instruction, 6-6,
9-48

move to processor register (MTPR)
instruction, 6-15, 7-22, 9-48
interrupt priority level register and,
8-21

interrupts forced by, 8-3
processor registers and, 3-7, 7-23
process space address translation
and, 7-14, 7-16

for software interrupt summary
register, 8-20

stack pointers referenced by, 3-6,
8-33

to write to map enable register,
7-19

move translated characters (MOVTC)
instruction, 6-2, 9-49

move translated until character
(MOVTUC) instruction, 6-2, 9-49

move zero-extended (MOVZ)
instruction, 6-14, 9-50

multiple register instructions, 6-14

multiply (MUL) instruction, 6-2, 6-11,
6-12, 9-50—9-51

multiply packed (MULP) instruction,
6-7, 9-51

multiprocessor systems
interrupt priority levels in, 8-18
interrupt requests in, 8-2
page table entries and, 7-11

multiprogramming, 2-1, 7-1
context switching in, 2-2—2-3
virtual memory in, 7-2

N

negative condition code, 2-14

next interval count register, 3-8

nibbles, 4-16, 7-3

notation conventions, 5-1—5-5

null strings, 4-1

numeric string data types, 2-12,
4-10—4-15

O

octaword data type, 4-9, 4-25

octawords, 7-3

opcode reserved to Digital fault, 8-12

opcode reserved to users fault, 8-12

operand processing, 2-7—2-8

operand reference exceptions,
8-12—8-14

operands
in instructions, 5-6, 5-9—5-10
in MACRO source statements, 5-5
notation conventions for, 5-1—5-2
in PDP-11 compatibility mode, 11-2

operating system, 7-2, 7-4, 7-6
memory management tables
controlled by, 7-7
page table entries changed by, 7-11
stacks used by, 8-28

operation codes (opcodes), 2-6

operation notation, 5-2—5-5

operators
in edit instruction, 9-28—9-34
in instructions, 5-6, 5-8
in MACRO source statements, 5-5
notation convention for, 5-4

options
compatibility mode instruction set,
10-2
floating-point instructions,
10-1—10-2
instruction emulation, 10-4
kernel instruction set, 10-3—10-4
MicroVAX I and II systems, 10-4
PDP-11 compatibility mode as, 11-1
processor registers, 10-2—10-3
string instructions, 10-2

OR instruction, exclusive OR (XOR),
9-34

outputs, edit instructions for,
6-9—6-10

overflow condition code, 2-14, 8-5

overflow exceptions, 8-4

overflows, 6-8

P

P0 Base Register (P0BR), 7-14

P0 Length Register (P0LR), 7-14

P0LR (length register), 7-9

- P0 page table (P0PT), 7-14
- P0PT (process space page table), 7-6
- P0 space, 7-2, 7-7, 7-13
- P1 Base Register (P1BR), 7-15
- P1 Length Register (P1LR), 7-15
- P1LR (length register), 7-9
- P1 page table (P1PT), 7-15
- P1PT (process space page table), 7-6
- P1 space, 7-2, 7-7, 7-13
- packed decimal data type, 2-12
 - edit instruction to convert to character string, 6-9
 - instructions for, 6-6—6-9
- packed decimal string data type, 4-16—4-17
- page frame number (PFN), 7-9, 7-10
- page mapping registers, 2-4
- pages (memory), 2-4, 7-1, 7-4, 7-6
 - access control for, 7-16
 - in PDP-11 compatibility mode, 11-8
- page table entry (PTE), 7-5—7-6, 7-8—7-10
 - changing, 7-11
 - for input/output devices, 7-10—7-11
- page tables, 2-4—2-5, 7-1, 7-4, 7-8
 - faults for, 7-20
- paging, 7-1
- parameters, for memory management, 7-20—7-21
- PDP-11 compatibility mode, 1-3, 8-7, 11-1
 - entering and leaving, 11-6—11-7
 - exceptions and interrupts in, 11-10
 - input/output references in, 11-11
 - memory management in, 11-7—11-9
 - processor registers in, 11-11
 - program synchronization in, 11-11
 - tracing in, 11-10
 - unimplemented PDP-11 traps in, 11-11
- PDP-11 compatibility mode, (*cont.*)
 - user environment emulation in, 11-2—11-6
- performance monitor enable register, 10-3
- peripheral device control/status and data (input/output) registers, 3-13
- peripherals
 - control, status and data registers in, 7-4
 - interrupts generated by, 8-19
 - page table entries for, 7-10—7-11
- physical addresses, 2-4
- polynomial evaluation (POLY)
 - instruction, 6-12, 9-51—9-52
- pop registers (POPR) instruction, 6-14, 9-52
- position-independent code, 2-6
- powerfail, 8-3, 8-6, 8-21
- power supply, for time-of-year clock, 3-7
- precision
 - of D__ floating-point data type, 4-3
 - of F__ floating-point data type, 4-4
 - of floating-point data types, 2-12
 - of G__ floating-point data type, 4-4
 - of H__ floating-point data type, 4-5
- priority dispatching, 2-3, 2-3
- priority levels
 - of exceptions and interrupts, 8-30—8-31
 - for interrupts, 3-18, 8-1—8-3, 8-18
 - for urgent interrupts, 8-21
- privileged instructions, 6-15—6-16
 - change mode, 9-14—9-15
 - extended function call, 9-35
 - halt, 9-37—9-38
 - load process context, 9-42
 - move from processor register, 9-45—9-47
 - move to processor register, 9-48
 - probe accessibility, 9-52—9-53

- privileged instructions, (*cont.*)
 - return from exception or interrupt, 9-56—9-57
 - save process context, 9-58—9-59
- privileged modes, 3-16, 7-1, 7-4
- privileged (processor) registers, 2-2, 3-7
 - asynchronous system traps and, 8-8
 - copies of process stack pointers in, 8-28
 - for memory management, 7-19
 - See also* processor registers
- privileged system services, 7-22—7-23
- probe accessibility instructions, 3-2, 6-15, 6-16, 7-22, 7-23, 9-52—9-53
- probe read accessibility (PROBER) instruction, 3-2, 6-16, 7-23, 9-52—9-53
- probe write accessibility (PROBEW) instruction, 3-2, 6-16, 7-23, 9-52—9-53
- procedures
 - call instructions for, 6-16—6-18
 - calls for, 2-7
 - return from procedure instruction for, 9-57
- process address space, 7-6
- process context, 3-2
 - interrupt stack during, 8-27
 - load process context instruction for, 9-42
 - save process context instruction for, 9-58—9-59
- process control block (PCB), 2-1—2-3, 3-6, 3-7, 3-9
- process control block base (PCBB) register, 3-9—3-13
- processes, 2-1—2-2
 - context switching for, 2-2—2-3
 - control instructions for, 2-8
 - multiprogramming execution of, 7-1
 - programming environment for, 3-1—3-4
 - virtual memory shared by, 7-2, 7-4
- processor errors, 3-18
- processor registers, 3-7—3-13, 10-2—10-3
 - move from processor register instruction for, 9-45—9-47
 - move to processor register instruction for, 9-48
 - in PDP-11 compatibility mode, 11-11
 - See also* privileged registers
- processors, 2-1—2-2
 - access modes for, 3-1—3-2
 - context switching in, 2-2—2-3
 - general registers in, 2-9
 - instruction operand processing by, 2-7—2-8
 - instruction set and, 2-5—2-6
 - interrupt requests arbitrated by, 8-2, 8-18
 - memory management and, 2-4—2-5
 - priority dispatching in, 2-3
 - process context on, 1-3
 - process control instructions in, 2-8
 - routine call capability in, 2-7
 - status during exceptions and interrupts of, 8-4—8-7
 - virtual addressing in, 2-4
- processor status longword (PSL), 3-1, 7-1
 - during exceptions and interrupts, 8-3—8-6
 - PDP-11 compatibility mode and, 11-7, 11-10
 - PDP-11 compatibility mode bit on, 11-6
 - status bits in, 8-29
 - trace exceptions and, 8-15
 - trace pending bit saved values in, 8-32
- processor status longword instructions, 6-18
 - bit clear processor status word, 9-5
 - bit set processor status word, 9-6
 - move from processor status longword, 9-47
- processor status word (PSW), 3-1, 8-4
 - bit clear processor status word instruction for, 9-5

processor status word, (*cont.*)
 bit set processor status word instruction for, 9-6
 PDP-11 compatibility mode and, 11-3, 11-7
 trap enable bits in, 6-17

process page tables, 2-5, 7-14

process scheduling interrupt, 8-3

process space, 2-4, 7-2, 7-7
 access control for, 7-17
 address translation for, 7-13—7-16

process space page tables, 7-6

program counter (PC), 2-9, 3-4
 addressing modes for, 5-36—5-41
 events and values saved in, 8-32—8-33
 during exceptions and interrupts, 8-3, 8-4
 not used in autodecrement mode, 5-13
 not used in register deferred mode, 5-33
 not used in register mode, 5-32, 5-33
 operands identified by, 2-6

program counter register addressing, 5-36—5-37
 absolute mode in, 5-37
 immediate mode in, 5-37—5-59
 relative deferred mode in, 5-41
 relative mode in, 5-39—5-40

programming
 trace exceptions during debugging in, 8-15
 use of stacks in, 3-14

programming environment, 3-1—3-4

programs
 stacks used by, 3-16
 virtual memory for, 1-3

program synchronization, in PDP-11 compatibility mode, 11-11

protection
 access control for, 7-16—7-19
 access modes for, 2-5

protection, (*cont.*)
 for memory, in multiprogramming, 7-1
 for page table entries, 7-9
 privileged instructions for, 6-16
 processor access modes for, 3-1—3-2

protection codes, 7-16, 7-18

push address (PUSHA) instruction, 6-1, 9-53

pushdown lists (stacks), 3-14

push longword (PUSHL) instruction, 6-13, 9-54

push registers (PUSHR) instruction, 6-14, 9-54

Q

quadword data type, 4-8—4-9, 4-24

quadwords, 7-3

queue data type, 2-13, 4-17—4-20

queue instructions, 6-19—6-24
 insert entry in queue, 9-38—9-39
 insert entry in queue at head, interlocked, 9-39—9-40
 insert entry in queue at tail, interlocked, 9-40
 remove entry from queue, 9-54—9-55
 remove entry from queue at head, interlocked, 9-55—9-56
 remove entry from queue at tail, interlocked, 9-56

queues, 4-17—4-18

R

range notation, 5-5

read access, 2-7

receive registers, 3-9

- register deferred index mode, 5-21—5-22
- register deferred mode, 5-33—5-35
- register mode, 5-32—5-33
- registers, 3-4
 - data in, 4-24—4-25
 - general, 3-4—3-6
 - general, addressing of, 2-9, 5-12—5-35
 - input/output, 3-13—3-14
 - in input/output device controllers, 2-15
 - interrupt priority level, 8-21—8-22
 - length, 7-9
 - for memory management, 7-19
 - multiple, instructions for, 6-14
 - page mapping, 2-4
 - for PDP-11 compatibility mode, 11-2—11-3
 - in peripheral devices, 7-4
 - pop registers instruction for, 9-52
 - privileged, 2-2
 - procedure calls and, 2-7
 - processor, 3-7—3-13, 10-2—10-3
 - processor, in PDP-11 compatibility mode, 11-11
 - processor status longword, 3-1, 7-1
 - program counter addressing modes, 5-36—5-41
 - push registers instruction for, 9-54
 - after restarts, 3-17
 - stack, 8-30
- relative deferred mode, 5-41
- relative mode, 5-39—5-40
- relative queues, 2-13
 - instructions for, 6-22—6-24
- remove entry from queue at head, interlocked (REMQHI) instruction, 6-24, 9-55—9-56
- remove entry from queue at tail, interlocked (REMQUI) instruction, 3-4, 6-24, 9-56
- remove entry from queue (REMQUE) instruction, 6-22, 9-54—9-55
- reserved addressing mode faults, 8-12
- reserved operand exceptions, 8-13—8-14
- restartability, 3-17—3-18
- return from exception or interrupt (REI) instruction, 2-14, 3-3, 6-16, 7-22, 9-56—9-57
 - asynchronous system traps and, 8-8
 - to enter PDP-11 compatibility mode, 11-6
 - interrupts triggered by, 8-3
 - program counter and processor status longword restored by, 8-4
 - service routines exit using, 7-23
 - trace exceptions and, 8-15
- return from procedure (RET) instruction, 9-57
- return from subroutine (RSB) instruction, 6-4, 9-58
- return instruction, 3-5, 6-16, 6-17
- rotate longword (ROTL) instruction, 6-14, 9-58
- rounded results, 6-11
- routines
 - call capability for, 2-7
 - case instructions for, 6-4
 - procedures, instructions for, 6-16—6-18
 - trace handlers, 8-17

S

- save process context (SVPCTX) instruction, 3-3, 6-15, 7-22, 7-23, 9-58—9-59
 - executed on kernel or interrupt stacks, 2-8
 - stack pointers referenced by, 8-28, 8-33
- scan characters (SCANC) instruction, 6-2, 9-59
- scheduling
 - interrupt for, 8-3
 - of processes, 2-8
- scratchpad registers, 3-7

- security
 - access control for, 7-16—7-19
 - access modes for, 2-5
 - privileged instructions for, 6-16
 - processor access modes for, 3-1—3-2
- self-relative queues, 2-13, 4-17, 4-18
 - instructions for, 6-22—6-24
- serious system failures, 8-14—8-15
- sharing of data, 3-3—3-4
- short literals, 5-29
- single-precision floating (F_—) data type, 4-3
- skip character (SKPC) instruction, 6-2, 9-59—9-60
- SLR (length register), 7-8
- software
 - asynchronous system traps and, 8-8—8-9
 - exceptions generated by, 8-2
 - interrupt priority levels for, 8-3
 - interrupts generated by, 8-19—8-21
 - memory management, 2-5, 7-1
 - process scheduling, 2-8
 - programming environment for, 3-1—3-4
 - registers available to, 3-5
 - system failures caused by, 8-14
- software context, 2-1
- software interrupt summary register (SISR), 8-19—8-20
- software process control block, 2-2
- span characters (SPANC) instruction, 6-2, 9-60
- special instructions, 3-2—3-3, 6-1
- stack frames (call frames), 2-7, 3-5, 6-14, 6-17
- stack pointers (SP), 2-13, 3-4, 3-6, 3-14, 3-15, 8-27—8-28
 - accessing, 8-30
 - not used in register mode, 5-33
 - for PDP-11 compatibility mode, 11-3
- stack registers, 8-30
- stacks, 2-13, 3-14—3-16, 8-27—8-30
 - autoincrement mode and, 5-14
 - during exceptions and interrupts, 8-3
 - in P1 space, 7-2
 - pop registers instruction for, 9-52
 - push registers instruction for, 9-54
- status bits, 8-29
- string instructions, 10-2
 - See also* character string instructions
- subroutines
 - branch to subroutine instruction for, 9-11
 - call instructions for, 6-4—6-5
 - calls for, 2-7
 - jump to subroutine instruction for, 9-41—9-42
 - return from subroutine instruction for, 9-58
- subscript range trap exceptions, 8-11
- subtract (SUB) instruction, 6-2, 6-11, 6-12, 9-60—9-61
- subtract one and branch (SOB) instruction, 6-4, 9-61
- subtract packed (SUBP) instruction, 6-7, 9-62
- subtract with carry (SBWC) instruction, 6-13, 9-62
- supervisor access mode, 2-5, 3-2, 7-1
- supervisor stack pointer (SSP), 8-27
- suspended instructions, 8-31
- swapping, 7-7
- switched-in context, 1-3
- switched-out context, 1-3
- synchronization
 - of access to shared data, 3-3—3-4
 - in PDP-11 compatibility mode, 11-11
- syntax, for operation of instructions, 5-2
- system address space, 7-6

system authorize database
 (SYSUAF.DAT), 7-22

system control block (SCB), 2-14,
 8-23—8-27

system control block base register, 2-14

system failures, 8-14—8-15

system identification (SID) register,
 3-13

system length register (SLR), 7-14

system maps, 7-20

system page table (SPT), 7-6

system page table entries, 7-20

system region page tables, 2-5

system services, privileged, 7-22—7-23

system space, 2-4, 7-2, 7-7, 7-8
 access control for, 7-17
 address translation for, 7-11—7-12

T

tail of the queue, 4-18, 6-19

test (TST) instruction, 6-12, 6-14, 9-63

time-of-year clock, 3-7

time-of-year block register, 10-3

trace bit, 8-6, 8-32

trace exceptions, 8-15—8-17

trace handlers, 8-17

tracing, in PDP-11 compatibility mode,
 11-10

trailing numeric string data type,
 4-12—4-15, 6-6

transfer instructions, 6-5

translation buffer invalidate all register
 (TBIA), 7-20

translation buffer invalidate single
 (TBIS) register, 7-20

translation buffers, 7-6, 7-20

translation not valid faults, 7-20, 8-12,
 8-32

translation of addresses, 2-4—2-5,
 7-8—7-16

transmit registers, 3-9

trap enable bits, 6-17, 6-18

trap-enable flags, 8-5

traps, 8-2
 asynchronous system, 2-3, 8-7—8-9
 unimplemented in PDP-11
 compatibility mode, 11-11

truncated results, 6-11

two's complement data representation,
 4-6

U

urgent interrupts, 8-21

user access mode, 2-5, 3-2, 7-1
 privileged system services accessed
 by, 7-22

user stack, 2-13

user stack pointer (USP), 8-27

V

variable-length bit field data type,
 2-12, 4-21—4-24

variable-length bit field instructions,
 6-25
 compare field, 9-17
 extract field, 9-36—9-37
 find first bit, 9-37
 insert field, 9-41

VAX-11/780 processors, 8-6

VAX systems
 MicroVAX I and II systems, 10-4
 PDP-11 compatibility mode on,
 11-1—11-11

vectors, in system control block,
 8-23—8-27

virtual address extension, 1-2—1-3
virtual address space, 1-4, 7-1,
7-6—7-8
virtual addresses, 7-1
virtual addressing, 2-4
virtual memory, 7-2—7-6

W

word data type, 4-7
words, 7-3
working memory, 7-2
write access, 2-7

X

XFC (extended function call)
instruction, 9-35
XOR (exclusive OR) instruction, 9-34

Z

zero condition code, 2-14, 8-5
zeros, 6-8, 6-10

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There is no handwriting or other markings on the paper.

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There is no handwriting or other markings on the paper.

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There are approximately 20 lines visible. The paper appears slightly aged or off-white. There is no handwriting or other markings on the page.

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There is no handwriting or other markings on the paper.

[illegible]

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

NOTES

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There is no handwriting or other markings on the paper.

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There is no handwriting or other markings on the paper.

VAX Architecture 1986

READER'S COMMENTS

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.) _____

What features are most useful? _____

Does the publication satisfy your needs? _____

What errors have you found? _____

Additional comments _____

Name _____

Title _____

Company _____

Dept. _____

Address _____

City _____

State _____

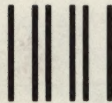
Zip _____

(staple here)

EB-26115-46

(staple here)

(please fold here)



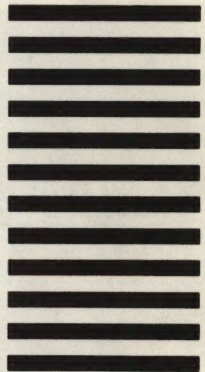
No Postage
Necessary if
Mailed in the
United States

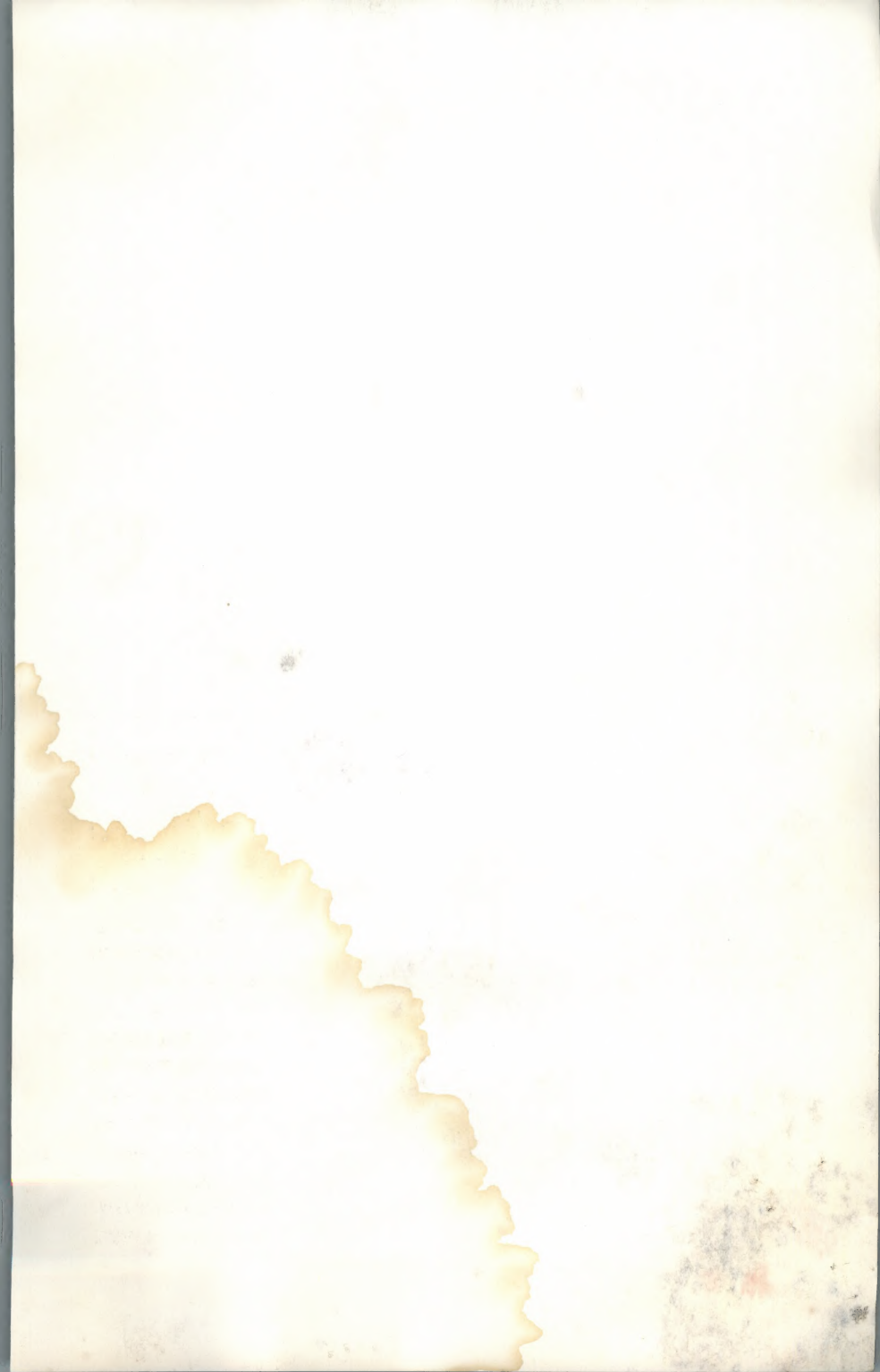
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Corporate Communications Group
CFO 1-2/M92
200 Baker Avenue
West Concord, MA 01742





digital